O D i M: A MODEL-DRIVEN APPROACH TO AGENT-BASED SIMULATION

Jaidermes Nebrijo Duarte and Juan de Lara
Escuela Politécnica Superior
Universidad Autónoma de Madrid, Spain
jaidermes.nebrijo@estudiante.uam.es, juan.delara@uam.es

KEYWORDS

Agent-Based Simulation, Model-Driven Development, Meta-Modelling, Domain Specific Visual Languages.

ABSTRACT

Model-Driven Software Development (MDD) is a software engineering paradigm that uses models as a means to specify, test, verify and generate code for the final application. Domain-Specific Visual Languages (DSVLs) are frequently used as high-level notations to specify such models, and hence the systems to be built. In this way, developers work with concepts close to the domain, and need not be experts in lower-level notations, thus increasing productivity.

In this paper we show the application of MDD to agent-based simulation. In particular we present the design of ODiM, a DSVL for Modelling and Simulation of Multi-Agent Systems. The language is made of four different diagram types, used to define agents' types, their behaviour, their sensors and actuators and the initial configuration. We have built a customized modelling environment integrated in Eclipse, and a code generator for MASON, a Java-based agent simulation language which allows the visual simulation of ODiM models.

INTRODUCTION

Model-Driven Development (MDD) is a software engineering paradigm which seeks increasing quality and productivity by considering models the core asset in software projects (Stahl and Volter 2006). They are used to specify, simulate, test, verify and generate code for the final application. Domain Specific Visual Languages (DSVLs) are graphical notations constrained to a particular domain and play a cornerstone role in MDD (Kelly and Tolvanen 2008). DSVLs provide high-level, powerful, domain-specific primitives, having the potential to increase productivity for the specific modelling task. Being so restrictive they are less error-prone than other general-purpose languages, and easier to learn, because the semantic gap between the user's mental model and the real one is smaller.

In computer modelling and simulation, systems are described using appropriate languages and notations for the purpose of virtual experimentation (Zeigler et al. 2000). Some of the disciplines that traditionally have used simulation as a research or decision-making tool include physics, mathematics, biology and economics. Lately, simulation has become popular in the social sciences with the emergence of agent-based simulation techniques (Gilbert, Troitzsch 1999). In this paradigm, the main concept is the agent (Jennings et al. 1998) (Wooldridge 1999), which can be defined as a "*computer system, situated in an environment, which is able to perform flexible and autonomous actions to achieve its design objectives*".

In agent-based modelling and simulation "*macroscopic*" phenomena emerge by the actions and interaction of elements at the "*microscopic*" level (Alfonseca and de Lara 2002) (Bonabeau et al 1999). Thus, the classical macro-level approach of representing systems as differential equations is no longer used. Instead, the variation of the quantities in the model is obtained from the explicit modelling of the individual elements. This presents several benefits, such as a deeper insight and understanding on the phenomena to be modelled.

Even though agent-based simulation is useful for a variety of disciplines, models are frequently described using low-level languages, often extensions of general purpose object-oriented programming languages like Java (Minar et al. 1996) (Luke et al. 2005). In this respect, our goal is to use MDD techniques to provide intuitive, visual notations for describing agent-based models in a precise way in order to facilitate the construction and execution of simulation models by non-programmers. For this purpose, we have designed a DSVL for modelling multi-agent systems called ODiM. The language is made of four different diagram types, used to define agents' types, their behaviour (based on state-machines), their sensors and actuators and the initial configuration. Thus, ODiM is tailored to agents of reactive type (Brooks 1991). We have built a customized graphical modelling environment integrated in Eclipse (Eclipse GMF 2009), and a code generator for MASON (Luke et al. 2005), a Java-based agent simulation language which allows the visual simulation of ODiM models. In this way, users of ODiM, just deal with a graphical notation, and do not need to be proficient in any low-level programming language.

Proceedings 23rd European Conference on Modelling and Simulation ©ECMS Javier Otamendi, Andrzej Bargiela, José Luis Montes, Luis Miguel Doncel Pedrera (Editors)
ISBN: 978-0-9553018-8-9 / ISBN: 978-0-9553018-9-6 (CD)

DOMAIN SPECIFIC VISUAL LANGUAGES

DSVLs are constrained notations tailored to specific domains and needs. MDD uses DSVLs in well-understood domains, where high-level, intuitive primitives can be identified to describe the problem at hand. Thus, the users of the DSVL are more productive as they are familiar with the primitives of the DSVL, the mental gap from the model to the problem is smaller, and no low-level, general purpose-languages are needed for encoding the solutions. In these restricted domains, code generators can be built to generate most of the code of the final system.

One way of specifying a DSVL is through a meta-model, which is model of the DSVL (abstract) syntax, and describes its main concepts. Meta-models are often specified using visual languages, such as UML class diagrams, and are usually complemented by textual constraint notations (like OCL) expressing additional language constraints. As an example, Figure 2 shows a part of the meta-model for the state machines that express the behaviour of ODiM agents. The meta-model describes that agents have behaviour, expressed through state machines. These contain a list of states, which can be normal, initial or final. States have transitions, which have a guard and an action. By means of OCL restrictions, we have specified that there is a unique initial state, which cannot receive incoming transitions, and that final states do not have outgoing transitions.

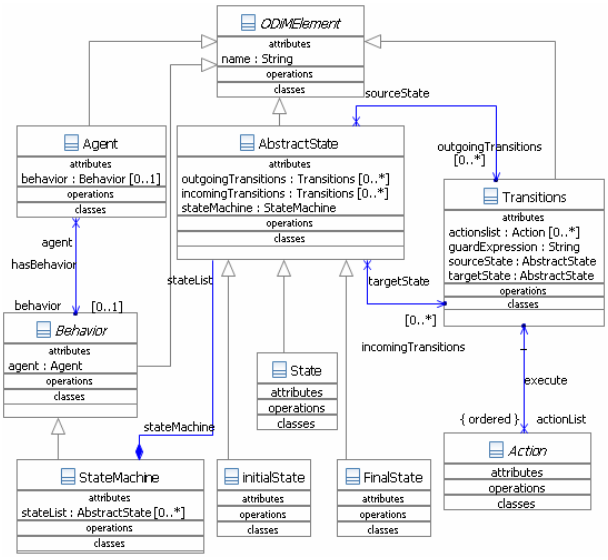

Figure 2: State Machine Meta-Model (partially shown).

The definition of a DSVL requires also a concrete syntax, specifying how each element is visually represented. In the simpler case, this is done by assigning graphical icons to classes in the meta-model and decorated arrows to associations. In the example of Figure 2, states are represented as ovals with the name inside, and transitions as edges. Figure 9 shows an example of a state machine. Meta-modelling tools like AToM^3 (de Lara and Vangheluwe, 2002), or GMF (Eclipse GMF 2009) are able to generate customized modelling environments for a DSVL given a meta-model and a description of its concrete syntax.

In addition to the syntax, it is necessary to specify the DSVL semantics, its meaning. One way to do this is to use a denotational style, relying in some other language for which the semantics are already defined. This is the approach we follow, by generating code for the MASON simulator (Luke et al. 2005).

Finally, DSVLs may also be used to describe complex systems that include different aspects or concerns. The complexity of these systems makes a common practice its specification by means of a set of smaller diagrams (called *views*) instead of including all the information in a single, monolithic one. Each one of these smaller diagrams is more comprehensible and cohesive, describing some feature of the system from a specific viewpoint. Thus, some DSVLs offer different diagram types in order to describe the various aspects of the system. These notations are known as Multi-View Visual Languages (MVVLs) (Guerra et al., 2005). For example, the UML notation proposes a set of diagram types to describe the static and dynamic aspects of the application. Note how, MVVLs are not described in a separate way, but the language is described through a common meta-model, which includes the meta-models of the different diagram types and their relation. In our case, the ODiM language is made of four different viewpoints to describe the agent structure, sensors and actuators, the agent behaviour and the initial configuration. The language is described in next section.

THE O D i M LANGUAGE

ODiM is a DSVL oriented to modelling and simulation with multi-agents. In a declarative way and through high-level visual notations, ODiM allows modellers to build simulation models based on the multi-agent approach. Thus conceptually, the main element of the language is the (embodied) *agent*, with the salient feature that every interaction has to occur through sensors and actuators.

We present the language through a simple simulation model, taken from (Miñano and Beltran, 2004). The model consists of a two-dimensional micro-world populated by agents, obstacles and goals. The micro-world is a multi-agent system where agents interact with their environment and other agents in order to reach long-term goals. Goals are static entities, sought by agents. They are called “long-term goals” because they are usually at a distance from agents, so it takes a certain amount of time for the agents to reach them. Goals have yellow colour, each one occupies one patch in the environment, and initially, they are scattered randomly. Obstacles are areas composed of contiguous

patches that cannot be occupied by agents and which agents cannot see through. They have black colour, and are also randomly distributed. As in ODiM interactions are embodied, goals and obstacles have an actuator to exhibit their positions in the micro-world as two-dimensional coordinates and the colour of their surface. This actuator generates a visual stimulus which can be perceived by the sensors of the agents.

Agents can move around and look for goals. Every agent can sense and act independently, based on a set of rules that neither explicitly indicate how to reach the goal nor provide a predefined sequence of steps to be followed to reach it; instead, it only tells the agent how to respond to obstacles and other entities in the environment. These rules are described in the state machine of each agent.

Agents have two-dimensional coordinates that specify their positions, and headings that indicate the direction of their movement. An agent can move one cell or patch per time unit in any direction relative to its current position (north, northeast, east, southeast, south, southwest, west, or northwest). An agent heading is defined as the angle between the linear path that links its positions at times `t-1` and `t` and the horizontal axis of the micro-world. Agent coordinates are initially set at random. Agents are also endowed with perception, which allows them to scan their neighbourhood in order to identify different kinds of entities (goals, obstacles, and other agents) and this is represented by a sensor. This sensor acts as the vision sense of agents and is designed to react to the visual stimulus generated by actuators of goals and obstacles. An agent state is defined by two values: (i) *Explorer*, meaning that the agent is looking for goals and then it is shown in green colour and (ii) *Rich*, when an agent reaches a goal, and it is shown in blue colour. Rich agents do not react to other entities, but stay in the position where they found a goal. When an agent detects a goal, it remembers its position and goes straightforward to it. If an obstacle is detected, the agent turns around 90º to avoid it. Finally, an agent changes its colour depending on what entity is detected: magenta when a goal has been detected, orange for a rich agent, brown for explorer agents and cyan when an obstacle has been detected.

Next subsections present the different ODiM diagrams, using the previous model as illustration.

The Agent's View

The agent view allows describing the internal structure of the agents. An ODiM agent is made of a name, basic properties and properties representing perception and action devices called *widget properties*. Basic properties are of some predefined primitive type such as *Boolean*, *String*, *Integer* and *Double*. Widget properties are sensors and actuators. These are devices allowing interaction between agents and their environment. This view also allows referring to the agent state machine, used to specify its behaviour. Figure 3 shows the meta-model of the agents' view. In ODiM the simulation environment is modelled as an agent, hence every model has initially an agent called *Environment*. As with all agents, we may associate a state machine to the environment agent and define a specific behaviour for it.

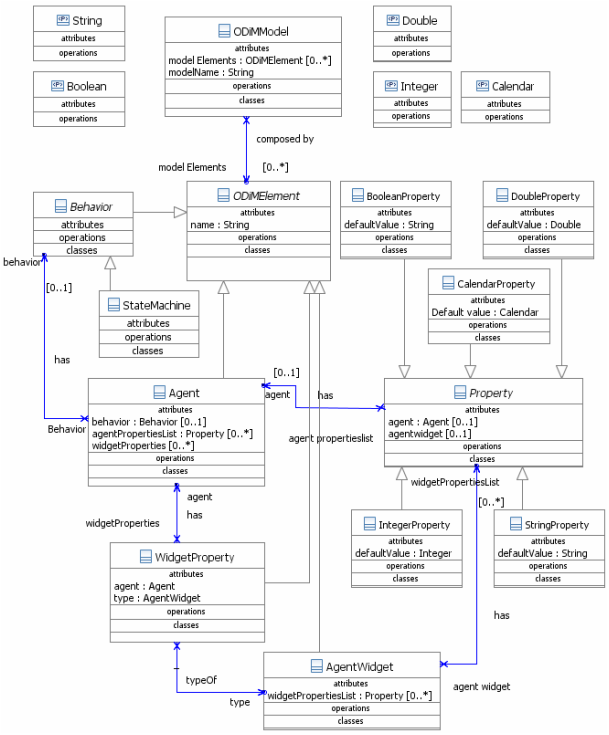

Figure 3: The Agents Viewpoint Meta-Model.

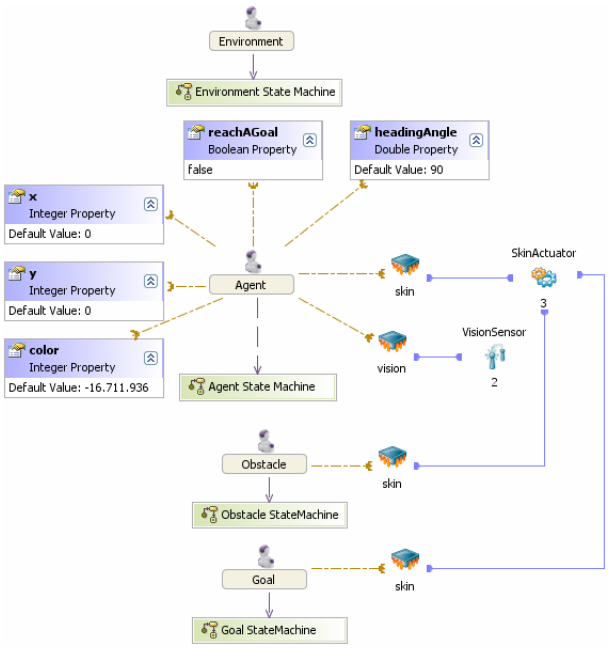

Figure 4: The Micro-world Agents in O D i M.

Figure 4 illustrates the agent structure of the example. The agent *Agent* has properties *x* and *y* that define its position. Property *color* is encoded as an integer with the RGB value. Property *reachAGoal* is used to manage the status of the agent. Initially, its value is false, but when an agent reaches a goal its value is set to true. Property *headingAngle* indicates the direction of movements and its initial value is 90º (i.e., north).

Sensors and Actuators: The Widget View

Inspired by the biological models of perception and communication, the interaction between agents and its environment is defined through the generation of stimuli and the response of the agents and the environment to those stimuli. An agent interacts with its environment and other agents through actuators and sensors that we call *widgets*. The *widgets view* describes the structure of actuators and sensors, the stimulus that can be generated by an actuator and the events produced by sensors as a reaction to the stimulus perception in the environment. Figure 5 shows its meta-model.

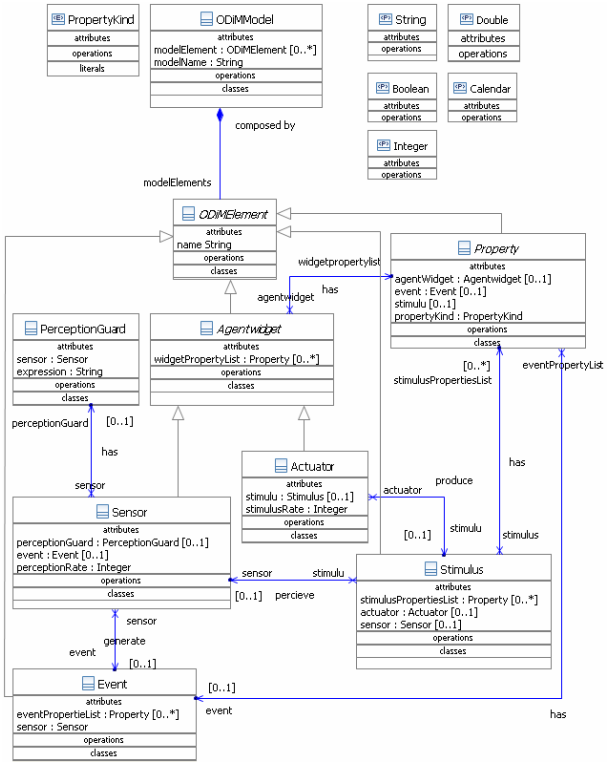

Figure 5: The Widget Viewpoint Meta-Model.

Actuators. An actuator is a kind of device that exhibits properties by generating stimulus. These properties may be characteristic of the actuator or internal agent properties. The *Stimulation Rate* in class *Actuator* is the frequency with which a stimulus is generated and propagated into the environment by an actuator. The frequency can be *on demand*, *permanent* or *on time intervals*. In the first case, the generation of stimulus takes place by an explicit invocation of a primitive action in the actuator. The execution of this action creates an instance of the stimulus, which will be added to the stimulus queue of the simulation environment. Once into the environment, the stimulus is available to be perceived by the rest of the agents or the environment itself. The second case (permanent stimulation) refers to the constant generation of stimulus by the actuator, at each simulation time. Finally, in the third case (on time intervals), the stimulus are generated by the actuator every certain number of time steps.

Each time an actuator generates a stimulus, the values of the properties that the actuator exposes are copied to it. Figure 6 illustrates the concrete syntax of an actuator in ODiM. The actuator *SkinActuator* is designed to expose the position of the agent in two dimensions (x,y coordinates), as well as its colour, encoded as an integer value. The information is exposed through a *VisualStimulus*. The frequency generation of the stimulus by the actuator is on time intervals because *stimulusRate = 3*.

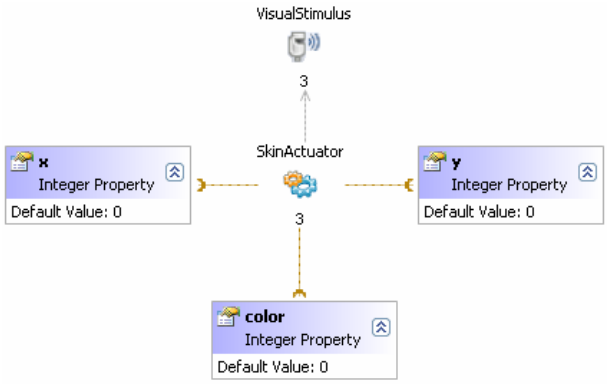

Figure 6: SkinActuator for Micro-world Example

Stimulus. A stimulus is a sensorial phenomenon immersed in the simulation environment and external to the agents. It represents a perceptible change in the environment. A stimulus may influence the behaviour of the agents and their environment, depending on the perception sensitivity of its sensors to that stimulus, and the interpretation and response of the agents. A stimulus in ODiM has a an intensity, which is the time during which it is available in the environment so that it can be perceived by all agents with sensors equipped with sensitivity to the stimulus. A stimulus can have permanent or limited intensity. In the first case, the stimulus is stored in the stimulus queue of the environment and is available at all times in the simulation. In the second case, the stimulus is available in the environment during the time of the simulation specified by the value of the intensity property.

Figure 7 illustrates the concrete syntax of a stimulus. The stimulus *VisualStimulus* is generated on time intervals by the actuator *SkinActuator* and it can be perceived for all agents equipped with sensors of type

VisionSensor, with a perception frequency of 2 simulation time units. Once the stimulus *VisualStimulus* has been generated, it will be available in the simulation environment during the next 3 simulation time units.

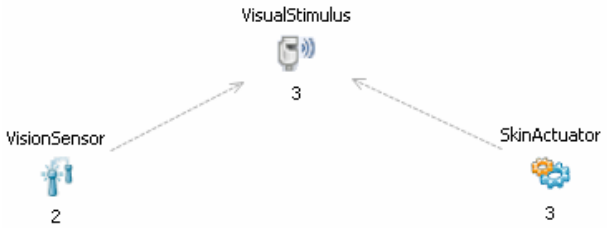

Figure 7: *VisualStimulus* in the Micro-world Example.

Sensors. A sensor is a device that provides agents with perception capabilities, allowing them to perceive stimulus immersed in the environment, generated by actuators of other agents. A sensor is composed of properties that define its internal structure. A sensor perceives stimulus according to a sampling rate called *Perception Rate*. A sensor can be designed so that it is sensible to stimuli under certain circumstances, which is expressed through a *Perception Guard*. This is a boolean expression representing a condition that must be met for that sensor to perceive the stimuli to which its capacity of perception was designed. The expression may contain references to the properties of the sensor, to the properties of the type of stimulus that the sensor can perceive, language primitive functions and constant values for primitive types in the language. Each time a sensor perceives a stimulus, it notifies the agent through the generation of events. These act as a mechanism for communicating information from the outside to the agent controller, represented by the state machine.

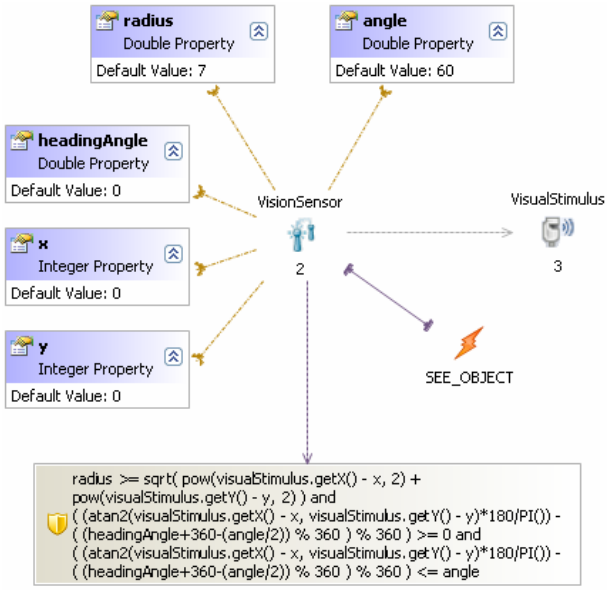

Figure 8: *VisionSensor* in the Micro-world Example.

Similar to actuators, sensors can have an on demand perception, permanent or based on a time interval. Figure 8 shows the definition of the sensor *VisionSensor* designed for visual perception, that is, to detect stimuli of type *VisualStimulus*. Properties *x*, *y* and headingAngle in the sensor are fed with the position and heading angle by the owner agent. When the guard expression is evaluated to true for some stimulus instance in the environment, then the sensor generates an event of type *SEE_OBJECT*. The event is sent to the internal event queue of the agent owner of the sensor and is available for the transitions in the state machine. Notice that the expression in the perception guard has nothing to do with any complex programming language. It just has mathematical terms and references to the properties of the sensor and the stimulus it perceives.

The Behaviour's View

The behaviour of agents is based on a reactive architecture (Brooks, 1991). The agents react to stimuli received through its sensors and interact with other agents and its environment through its actuators. The only information available to agents is an internal queue of events, produced by sensors. These events are no more than a copy of the information available in the environment, which is propagated through the stimuli generated by other agents or by the environment. The behaviour of an agent is defined visually using a state machine. This represents a single and independent thread of control, which determines the agent internal state in an instant of the simulation. Figure 9 shows the state machine for the agents of the example.

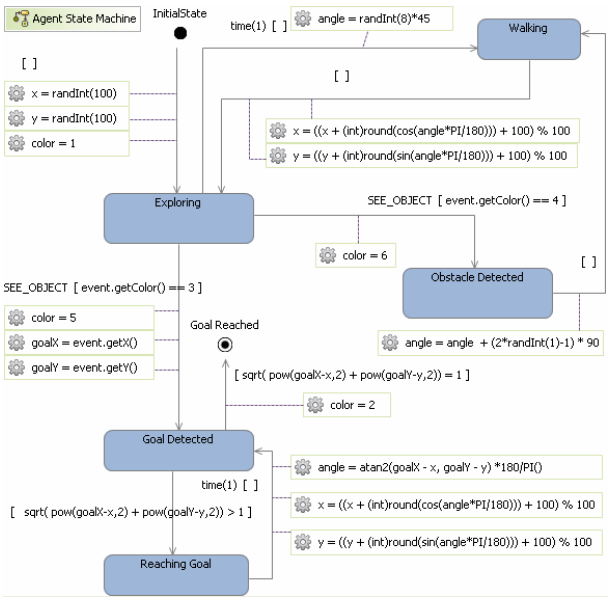

Figure 9: State Machine for the Micro-world Agents.

The agent state machine has a high dependency on sensors and actuators. It can modify the agent's internal state, made of the values of each one of its properties, the state of its sensors and actuators, and the state of the queue of events generated by each one of its sensors.

The state of the sensors and actuators is determined by the values of each one of its properties. The properties of an agent and its sensors and actuators are visible to the states machine of the agent. An agent can modify the properties of its sensors and actuators, but those will not be able to change the properties of the agent.

The transitions occur instantaneously and modify the current state of the machine moving it between states. A transition is enabled if the following conditions hold: (i) the source state of the transition is the current state on the state machine, (ii) the trigger event of the transition has been generated by any sensor and is available in the internal event queue of the agent, and (iii) the guard of the transition is evaluated to true. Once a transition is enabled, it is fired, leading to the execution of the associated actions and a change of state. If several transitions are active, we use the priority of the sensors that have generated the events, choosing the transition whose trigger was generated by the sensor with higher priority. If there are multiple sensors with the same priority, a transition is chosen randomly.

In ODiM, transitions may include time events (with syntax `time(t)`). Similar to DEVS (Zeigler et al. 2000), time events determine how much time the agent must wait in a state in the absence of external events. The time can be a constant or be defined by a property.

The Instances View

This view allows the configuration of the initial state of the system (e.g. positions of agents, obstacles and goals, etc). By space limitations, we omit its description.

THE MODELLING ENVIRONMENT

We have built a modelling environment for ODiM, and integrated it in the Eclipse framework. For this purpose, we have used the GMF (Eclipse GMF 2009) toolkit, and designed a MVVL with four different views. Figure 10 shows a screenshot of the environment.

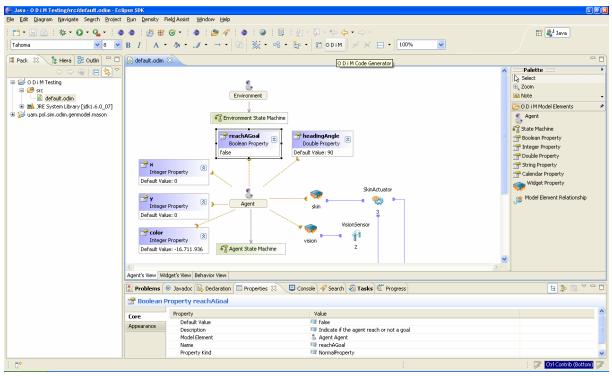

Figure 10: Modelling Environment for O D i M.

Code Generation for MASON

MASON (Luke et al. 2005) is a single-process, discrete event simulation core and visualization library written in Java, designed to be flexible to be used for a wide range of simple simulations, but with a special emphasis on swarm multi-agent simulations. In MASON, agents are “brains” which may or may not be embodied. In the latter case, we call them embodied agents. Note that, as the simulation designer uses ODiM (where interactions are embodied), we force MASON agents to be embodied. MASON does not schedule events to send to agents; instead, it schedules the agent to be stepped (pulsed or called) at some time in the future. Hence, MASON agents implement the *Steppable* interface.

MASON can schedule Steppable objects to occur at any real-valued time in the future. However, we schedule the agents in ODiM in a discrete way. The Schedule may be divided into multiple orderings that further subdivide a given time step: agents scheduled at a given time but in an earlier ordering will be stepped prior to agents scheduled at the same time but in a later ordering.

MASON is written in three layers: the utility layer, the model layer, and the visualization layer. The former consists of classes that may be used for any purpose, e.g. random-number generators, data structures and other facilities. The model layer is a small collection of classes consisting of a discrete event schedule, schedule utilities, and a variety of fields that hold objects and associate them with locations. This code alone is sufficient to write basic simulations running on the command line. The visualization layer permits GUI-based visualization and manipulation of the model. For most elements in the model layer, down to individual fine-grained objects in the model, there is an equivalent “proxy” element in the visualization layer responsible for manipulating the model object, portraying it on-screen, and inspecting its contents.

Code generation techniques save time in projects, reducing the amount of tedious redundant programming. Code generators are powerful tools, but they may become very complex and hard to understand. One way to reduce complexity and increase readability of model compilers is to use *templates* for code generation. Java Emitter Templates (JET) (Eclipse EMF, 2009) is a tool provided by The Eclipse Modelling Framework (EMF) project that allows using JSP-like syntax to write templates expressing the code that needs to be generated. A JET template is a text file, from which JET generates a set of classes that contains the code needed for the generation. The JETEmitter class provides a convenient high-level API to transform templates and generate code programmatically. The *generate* method of the JETEmitter class combines template translation and text generation into a single step.

The strategy we have followed for generating MASON code is to build a set of JET templates that collect rules for translation between the ODiM simulation model and the programming model of MASON, combined with a framework (utility classes) built on MASON that supports its programming model. The supporting framework consists of a set of utility classes (*AbstractAgent, AbstractSensor, AbstractActuator, AbstractStimulus, etc*) that abstract the common operations for the main concepts in ODiM. As MASON agents are modelled by a Java class, implementing the interface Steppable, those classes are implementations of the class Steppable. The advantage of our strategy is that JET templates are simpler and easier to maintain than a specific, hard-coded model compiler.

Thus, agents, sensors, actuators, stimulus and the simulation environment are translated to MASON as classes which inherit from its corresponding abstract class in the supporting framework. For example, the agent *Agent* in the micro-world generates the class *MASONAgent* which extends from *AbstractAgent*. Basic properties are synthesized as private attributes whose data type has the proper correspondence with Java types. Widget properties are generated in the same way, but the attribute type is the corresponding class generated for the sensor o actuator. Stimulus and Events are Plain Old Java Objects (POJOS). In all cases, getter and setter methods are provided to have access to attributes.

The actions of the state machine of the agents are coded in the *step* method, which is executed at each step of the simulation according to the schedule of the agent. Instructions available for transitions in the state machine are supported by a set of language primitives as *scan*, *createAgent*, *generateStimulus*. Also, mathematical operations may be used.

Figure 11 illustrates an execution of the example, where we can see the adaptive behaviour of the agents to different entities inside the environment, according to the ODiM model.

RELATED WORK

Several agent-based languages can be found in the literature, like Swarm (Minar et al. 1996), Ascape (Inchiosa and Parker, 2002) and MASON (Luke et al. 2005). Most of them are libraries for object-oriented, textual programming languages. This requires the users to be proficient in programming tasks, and in addition, none of them is based on a graphical notation.

There have been some attempts to design graphical modelling notations for designing multi-agent systems. For example, the FIPA modelling committee is working on AUML, (Odell 2004). Although their aim is to be domain independent, in the area of simulation many domain-specific concepts are necessary (like sensors, actuators, stimuli, etc.). Thus, we believe creating a DSVL makes sense. Such DSVL has the goal of offering high-level, customized constructs for the task to be performed, as well as constraining the user as much as possible. Therefore, we believe that using a customized modelling notation for agent simulation, the user is more productive than using a more general one. Moreover, AUML is only a design notation, and thus the resulting specifications are not executable.

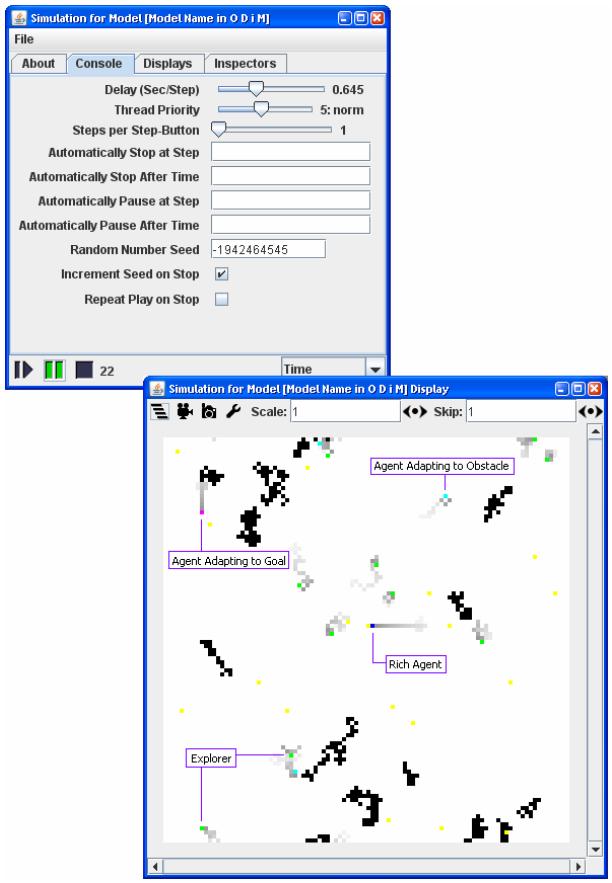

Figure 11: A step in the Simulation of the Example.

In (Muzy et al 2007), we developed a graphical notation for agents, where no code generator was provided, but only a mapping into DEVS (Zeigler et. al 2000).

CONCLUSIONS AND FUTURE WORK

In this paper, we have presented ODiM, a DSVL for agent-based modelling and simulation. The aim of the language is to provide high-level, visual constructs for building easily agent-based models that can be simulated by using MASON as a back-end. ODiM provides different diagrams to specify the different aspects of the simulation, thus making the design modular. We have built a customized modelling environment and a code generator for MASON, so that the user does not have to know any programming language.

There are many possible lines of research. We are currently working in providing three dimensional graphical output forms, in extending ODiM so as to allow the control of agents with neural networks, and in developing further code generators, for example to synthesize controllers for real robots (Lego, 2009). We are also working on including library concepts and reusing facilities, so as to allow reusing already defined sensors, actuators and controllers.

Acknowledgements: Work partially sponsored by the Spanish Ministry of Science and Innovation with project METEORIC (TIN 2008-02081).

REFERENCES

Alfonseca, M., de Lara, J. 2002. "Two level evolution of foraging agent communities". BioSystems Vol 66, Issues 1-2, pp.: 21-30. Elsevier.

Bonabeau, E., Dorigo, M., Theraulaz, G. 1999. Swarm intelligence: from natural to artificial systems. *New York. Oxford University Press.*

Brooks, R. A. (1991). Intelligence Without Representation, Artificial Intelligence 47, pp.: 139-159.

Eclipse (2009). The Eclipse Graphical Modelling Framework (GMF). http://www.eclipse.org/modeling/gmf/

Eclipse (2009). The Eclipse Modelling Framework (EMF). http://www.eclipse.org/modeling/emf/docs/

Gilbert, N., Troitzsch, K. 1999. Simulation for the Social Scientist. *Open University Press.*

Guerra, E., Díaz, P., de Lara, J. (2005). "*A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views*". Proc. IEEE VL/HCC'05, Dallas, pp. 284-286.

Inchiosa, M. E, Parker, M. T. 2002. "*Overcoming design and development challenges in agent-based modeling using ASCAPE*". Proc. of the National Academy of Sciences of the United States of America, Vol 99(3), pp.: 7304-7308.

Jennings, N.R., Sycara, K., Wooldridge, M. 1998. A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems, 1, pp.: 7-38. Kluwer.

Kelly, S., Tolvanen, J. P. 2008. Domain-Specific Modeling: Enabling Full Code Generation. *Wiley-IEEE Computer Society Press*.

de Lara, J., Vangheluwe, H. 2002. "*AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling*". Proc. FASE'02, LNCS 2306, pp.: 174-188. Springer.

Lego Mindstorms, 2009, http://mindstorms.lego.com/

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. 2005. "*MASON: A Multiagent Simulation Environment*". SIMULATION, Vol. 81, No. 7, 517-527.

Minar, N., R. Burkhart, C. Langton, and M. Askenazi. 1996. The Swarm simulation system: a toolkit for building multi-agent simulations. Working Paper 96-06-042, Santa Fe Institute. See also: http://www.swarm.org.

Miñano, M., Beltran, F.S. 2004. "*Reaching long-term goals based on local interaction between the organism and its environment: computer simulations based on adaptive behavior*". Perceptual and Motor Skills 99, 27-33.

Muzy A., de Lara, J., Guerra, E. "*Designing PRIMA: A Precise Visual Language for Modeling with Agents, in a Physical environment*". Proc. MSV 2007, pp.:231-238

Odell, J., Nodine, M., Levy, R. 2005. "*A Metamodel for Agents, Roles and Groups*". Proc. AOSE'04, LNCS 3382, pp.: 78-92. Springer. See also the AUML home page: http://www.auml.org.

Stahl, T., Völter, M. 2006. Model-Driven Software Development. *Wiley*.

Wooldridge, M. 1999. Intelligent Agents. In "Multiagent Systems. A modern approach to Distributed Artificial Intelligence" (Weiss ed.). pp. 27-77, The MIT Press.

Zeigler, B. P., Praehofer, H., and Kim, T. G. 2000. Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. *Academic Press*

BIOGRAPHIES

Jaidermes Nebrijo Duarte is a PhD student in Computer Science of the Escuela Politécnica Superior at the Universidad Autónoma de Madrid (Spain). He has nine years of experience in the area of software engineering. Currently he is working in the area of IT Consulting and Outsourcing as J2EE IT Architect in the framework of a project at IBM Global Services España for Mapfre Insurance.

Juan de Lara is associate professor at the Computer Science department of the Escuela Politécnica Superior at the Universidad Autónoma de Madrid (Spain), where he teaches Software Engineering and Automata Theory. His research areas include Model-Driven Development, Visual Languages, Model Transformation and Simulation. He has published more than 80 papers in international journals and conferences.