Designing PRIMA: A Precise Visual Language for Modeling with Agents, in a Physical environment

Alexandre MUZY
CNRS-LISA
Università di Corsica – Pasquale Paoli (France)
a.muzy@univ-corse.fr

Juan de LARA
Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)
jdelara@uam.es

Esther GUERRA
Universidad Carlos III de Madrid (Spain)
eguerra@inf.uc3m.es

ABSTRACT
In this paper we introduce PRIMA: A PRecise vIsual language for Modeling with Agents, in a physical environment. Our objective is to design an expressive Domain Specific Visual Language for building agent-based models. Models in PRIMA are diagrams, which describe the agent structure (i.e. actuators and sensors), behavior, instances and environment topology. We show the use of metamodeling techniques to describe the multi-view language PRIMA, its definition in the AToM^3 tool, and the generation of a customized modeling environment for it.

Keywords
Domain Specific Visual Languages, Metamodeling, Agent-Based Simulation, AToM^3.

1. INTRODUCTION
Domain Specific Visual Languages (DSVLs) are high-level, visual notations especially designed for modeling in particular domains. They provide intuitive, powerful constructs and abstractions for performing activities in an efficient way. The high-level of abstraction of DSVLs frees the user from dealing with low-level, *accidental* details. The language we present in this paper is one of such DSVLs.

A good way to deal with the ever increasing complexity of systems is to divide their specification into diagrams, each dealing with a concrete aspect of the system. These smaller diagrams are simpler, more comprehensible and cohesive than a bigger, monolithic one. Thus, some DSVLs are made of a set of diagram types, each one of them especially designed for the description of the system under a specific point of view. We call such languages Multi-View Visual Languages (MVVLs) [8, 9]. A prominent example of MVVL is UML 2.0, which provides different diagram types to describe the static structure, interactions, activities and states of the system [6].

The DSVL we propose in this paper is oriented to the area of computer simulation [21]. This is an inexpensive means to perform virtual experiments that would be difficult, non-ethical, expensive or impossible to perform in reality. Many disciplines – such as physics, mathematics, computer science, biology and economics – use simulation techniques as a means of research or decision making. With the emergence of agent-based simulation techniques, simulation is also becoming popular in the social sciences. In agent-based modeling and simulation, the fundamental element is the agent [10, 13, 19, 20], 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 this way, the simulation is performed by the interactions of the agents, which give rise to macroscopic or emergent phenomena [1].

In many application areas, agent-based models are built by interdisciplinary teams which may or may not include computer scientists. Many agent-based simulation languages have been proposed in the literature [1, 2, 19]. However, most of these languages are low-level or even based on extensions of object-oriented programming languages such as Java. This makes the agent-based languages difficult to use for non-computer scientists. While these frameworks are used for *programming*, in contrast, what we propose here is a high-level, visual language for *agent-based modeling* from which code can be synthesized[1]. That is, our aim is to introduce Model Driven Development (MDD) concepts in the area of computer simulation [6].

The aim of this work is to define a high-level MVVL to enable modelers to deal with agent-based simulation models through visual constructions. Contributions of this paper are: (i) the definition of a high-level, formal MVVL for agent-based modeling, and (ii) the use of novel metamodeling techniques for the specification of such language with the AToM^3 tool [11].

Paper Organization. Section 2 presents a brief overview of agent-based simulation. Section 3 presents our metamodeling approach to handle MVVLs. In section 4, we show the definition of PRIMA. Section 5 sums up the specification in AToM^3. Section 6 describes simulation structures. Section 7 gives an example, showing the generated modeling environment with AToM^3. In section 8, the work is situated in comparison to related work. Finally, conclusions and perspectives of this research are given in section 9.

2. AGENT-BASED SIMULATION

In agent-based simulation, the key concept is the *agent*. An agent is situated in an environment, from which it receives input stimuli (by means of sensors) and to which produces actions (by means of actuators). In addition, an agent has some objectives

[1] Obviously, modelling is not reduced entirely to graphical specifications. Precise codes can be added when required.

which it tries to satisfy with his behavior. Several approaches can be found in the literature to describe agent behavior [10]: logic-based architectures, Belief-Desire-Intention (BDI) architectures and reactive and layered architectures. In the first approach, the agent has a model of the environment based on first-order logic, and decides the action to be taken based on inference rules. The BDI approach is based on practical reasoning. The agent has databases of beliefs, desires and intentions. The latter can be satisfied by means of plans of atomic actions. Finally, reactive architectures propose a direct mapping from sensors to actuators, where the agent does not store models of the environment.

In agent-based simulation, space (i.e. the environment) is often discretized. Moreover, frequently the environment is not passive, but plays a certain role (*i.e.*, is assigned some properties that evolve over time). For example, in a simulation of a fire spread, the environment performs the calculation of the fire front. Sometimes cellular models [15] are used for the representation of the environment. In agent-based simulation, agents are placed in cells and may be able to move through them.

In the PRIMA language we propose specifying agent behaviors using state automata, whose transitions may be fired by events coming from sensors, which may invoke events in actuators and sensors. Agents can interact with other agents or the physical environment. Using state automata behaviors can be specified freely by states and transitions. Interactions of agents with agents and space are graphically and textually described by the language. Choosing such architecture lets free the modeler to deal with BDI and logic-based architectures. This reactive architecture tackles the important aspects in modeling agents in a physical environment. Adding precise higher-level structures for social interaction is up to future work.

3. METAMODELING MVVLs

We use metamodeling as a means to describe the abstract syntax of Visual Languages (VLs). In this case, a metamodel is used to describe all the admissible models of the language (i.e. those models conformant to the metamodel). Typically, this is specified by means of class or Entity-Relationship diagrams. Frequently, metamodels are provided with additional constraints, possibly expressed in some textual constraint language (such as OCL). Thus, the specification of the VL relies in a purely declarative approach (the metamodel), combined with a procedural part based on constraint checking. For the definition of the concrete syntax (*i.e.*, how the abstract syntax concepts have to be rendered), if the structure of the visualization is similar to the abstract syntax, we can just assign a graphical representation to each element of the abstract syntax.

The metamodeling tool AToM^3 [11] allows defining DSVLs by means of metamodeling and manipulating models by means of graph transformation [5]. The latter is a visual and formal technique to rewrite graphs based on rules. Graph transformation is based on defining graph-grammars. A graph-grammar is a set of rules. Each rule has a left and a right hand side (LHS and RHS) containing graphs. When applying a rule to a graph, first an occurrence of the LHS has to be found in the graph, and then it can be substituted by the RHS. The theory of graph transformation has been lifted to rewrite more complex structures such as triple graphs [5, 7, 18]. These are made of two graphs (called source and target) related through a correspondence graph. Nodes of the latter have morphisms to elements of source and target graphs. Grammars rewriting triple graphs are called Triple Graph Grammars (TGGs).

MVVL models are composed of a set of diagram types. In the metamodeling approach, the definition of each one can be based on a unique metamodel, which relates their abstract syntax elements. UML2.0 illustrates such an approach. The different diagram definitions, that we call *viewpoints* may have some overlapping parts in this unique metamodel.

At the model level, to guarantee syntactic consistency, we build a unique model (called *repository*) by gluing all system views built by the user. The resulting model is conformant to the complete VL metamodel. This gluing operation is performed by automatically generated TGG rules derived from the metamodel information (see for example [8] for a detailed description). Updating the repository as a result of a system view modification may leave some other views in an inconsistent state. At this point, other automatically generated triple rules update the necessary system views. In this way, we have TGGs that propagate changes from the system views to the repository and the other way around. This mechanism is similar to the model-view-controller framework.

Although this consistency mechanism could have been hard-coded in the generated environment; we believe it is interesting to explicitly generate TGG rules for it. The advantage is that their graphical nature allows the user to easily understand and modify them, allowing a customization of the MVVL environment behavior. Moreover, it is possible to generate different sets of rules to model different behaviors for the environment.

For additional domain specific syntactic constraints, the MVVL designer may provide additional TGG rules. For dynamic semantics consistency, the designer must provide the necessary mechanisms. Usually, these will involve checking at the repository level, probably performing a model transformation into another domain for analysis or simulation, but other specific checking can be provided as well.

4. DEFINITION OF PRIMA

Separating structure from behavior permits simplifying the decomposition and the specification of the internal structure of a system. In PRIMA, viewpoints are combined to compose a multi-agent system through: (i) its topology (cell space topology[2], instances, interactions between agents and environment), and (ii) its behavior. Complexity of multi-agent systems emerges from the number of interactions and sub-components constituting them. Using different views is necessary and adequate to describe precisely every part of the whole system. All these viewpoints have been implemented in AToM^3 and we briefly describe them in the following subsections.

4.1 Cellular Space Topology

The *Topology* viewpoint is used to describe cellular spaces (*i.e.*, the agent environment). As restricted to two dimensions, a cellular space is considered as a square/rectangle of interconnected cells. Thus, as Figure 1 shows, the main class is the *CellularSpace*, which has as attributes the style of the horizontal and vertical borders. The border style can be *circular,*

[2] A non-cellular based environment being merely a one-dimension cellular environment.

to denote that the border is connected with the opposite border; *fixed_value*, to denote that border cells have a fixed value; *connected*, when border cells are connected to cells of other cellular spaces; and *reflective*, when the left (resp. right, upper, bottom) boundary cells are kept identical to the most left (resp. right, upper, bottom) normal cells. A cellular space is made of *Cells*, which are assigned physical dimensions (*sizeX* and *sizeY*).

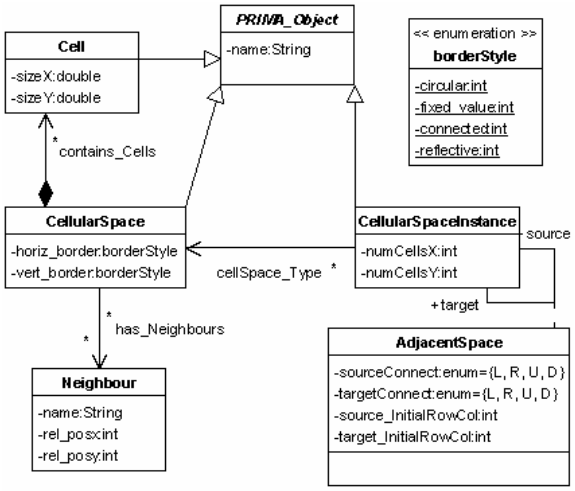

Figure 1: Topology Viewpoint.

A cellular space can be assigned a particular neighborhood for all its cells. This is specified by assigning *Neighbour* objects to the cellular space. These are used to specify a neighbor name and relative position for an arbitrary cell. For example, we can define "N" (north) as the name of the neighbor which is at relative coordinates (0, -1). There are several popular neighborhoods, such as *von Neumann* (a cell has 5 neighbors: north, south, east and west, as well as itself) and *Moore* (a cell has 9 neighbors, the first 5 as in von Neumann, and then north-east, north-west, south-east, and south-west).

Cellular spaces may have instances (*CellularSpaceInstance*), which have as attributes the number of cells in the horizontal (*numCellsX*) and vertical dimensions (*numCellsY*). Instances can be connected through relation *AdjacentSpace* to form more complex geometries. An adjacent space has attributes to specify source borders and target cellular space instances that are being connected (*sourceConnect* and *targetConnect*), as well as the index of the first cell that is connected in the source and target borders (*source_InitialRowCol* and *target_InitialRowCol*).

4.2 Interactions

The *Interactions* viewpoint describes agent structures (*i.e.*, their actuators, sensors and properties), as well as how they can interact each others and with the environment. This viewpoint is shown in Figures 2 and 3. Figure 2 shows how the environment (*World*) is made of objects (*PRIMA_Object*) with certain properties (*Property*). There are properties with predefined types such as *int*, *double* or *string* (not all shown in the figure), which have an attribute to be assigned an initial value.

This viewpoint also defines *Regions*, which are active sets of cells corresponding to different locations in space. Members of regions are updated dynamically depending on a condition. Regions can invoke events to be sent to cells or agents (see Figure 3) when these conditions are met. A cell contains physical objects, which can be opaque (*e.g.*, to control if a vision sensor can see through), or not visible (*e.g.*, for an agent corresponding to an institution).

Class *Cell*, defined in the *Topology* viewpoint, is the one used in this viewpoint in order to assign properties and behaviors. Note however that in this viewpoint we do not need to show its attributes, but we need to identify cells by names. In this way, in the environment to be generated, cells referenced in this viewpoint will only show its name, but not attributes *sizeX* and *sizeY*. Note also that if a certain entity appears in different viewpoints, we can assign it different concrete syntaxes.

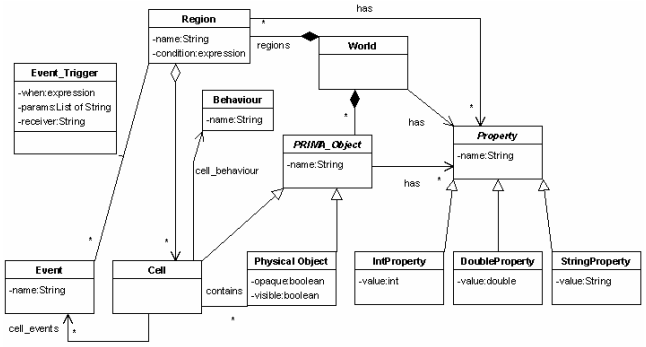

Figure 2: Interactions Viewpoint (partially shown)

Figure 3 shows another excerpt of the viewpoint, which shows how *Agents* are *Physical Objects* made of *Agent Widgets*. These can be either sensors or actuators. We have defined three types of sensors: *VisionSensor*, *PropertySensor* and *CommunicationSensor*. The first one is used to model a vision device in the agent. The property sensor is used to model a generic sensor, which may sense some property of the environment (*e.g.*, humidity, temperature). The communication sensor models a communication device, which may receive messages from a communication actuator. A message has a name (*i.e.*, its type), and a number of properties (*i.e.*, its data).

We identified three types of actuators: *CommunicationActuator*, *MovingActuator* and *PropertyActuator*. The first one is used to send a message to a receiver agent having a compatible communication sensor. The moving actuator is used for physically moving the agent through the environment. Finally, the property actuator models a generic actuator able to change some property of the environment (e.g. temperature, brightness).

Both cells and agents can be assigned a behavior, defined in the *Behavior* viewpoint. Agents and cells can also declare events, which can be invoked by regions. The agent (resp. cell) can react to these events as it has a state automaton (*Behavior* viewpoint) where actions can be performed on the occurrence of events.

Note how, we have included methods in the metamodel (we do not show the parameters for simplicity). This unusual decision is justified due the fact that in the behavior automaton it is possible to include references to sensor methods in the transition events, as well as invocation of actuator methods in the actions. In the execution platform, these methods are mapped onto the Discrete Event System Specification (DEVS) framework [21].

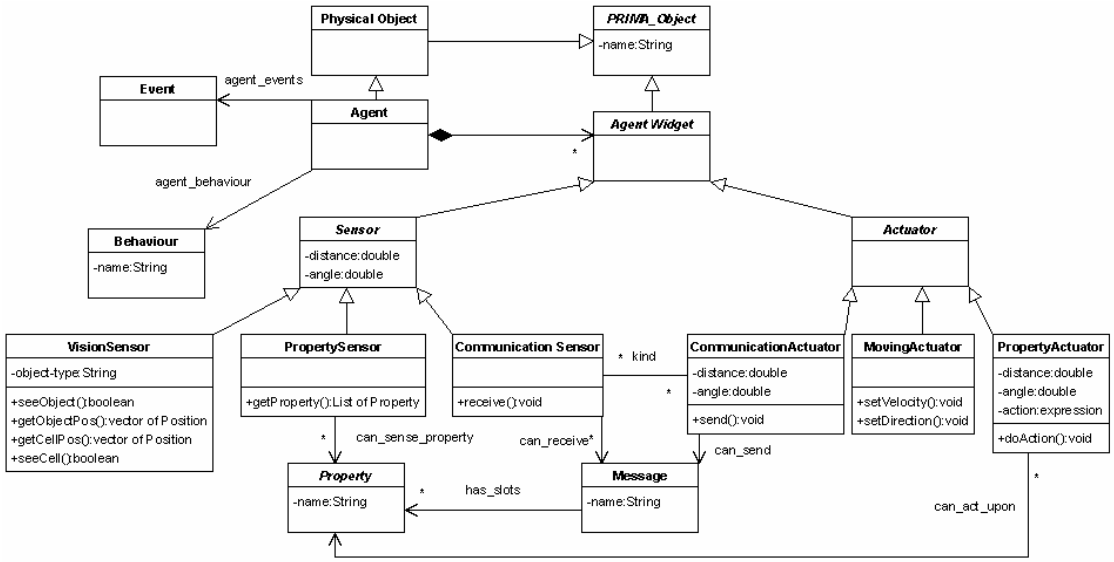

Figure 3: Interactions Viewpoint, second part

4.3 Instances

The *Instances* viewpoint allows defining the simulation initial state, as well as configuring initial values for properties of agent and cell instances. The viewpoint metamodel is shown in Figure 4. Class *AgentInstance* represents a set of agent instances whose cardinality is given by attribute *numberOfAgents*. Each agent instance is assigned a position *(x, y)* in the cellular space given by attribute *cellSpaceInstanceName*. In addition, agents have a direction (an orientation in the two dimensional space). It is possible to set the initial value of agent and cell properties by means of *PropertyInstance*. Although in the *Interactions* viewpoint we can assign an initial value for a given property, here we can initialize the value for particular agent and cell instances. Cell instances are initialized in a similar way as agent instances. They are referenced by an index and by the cellular space instance where they belong.

This viewpoint allows also configuring simulation experiment parameters. For example, for each property instance, we can set a quantum size, which is used when a continuous variable is discretized and a discrete event simulation approach is followed (*i.e.*, it is dual to the time advance interval). The user can also configure the simulator type by means of entity *Simulator*. This class allows setting the simulation method (discrete event or discrete time), the time advance interval *delta_t* (in case of a discrete time simulation), a condition to finish the simulation (it can depend on the simulation time or on model properties), and the integration method in case some expression has to be integrated.

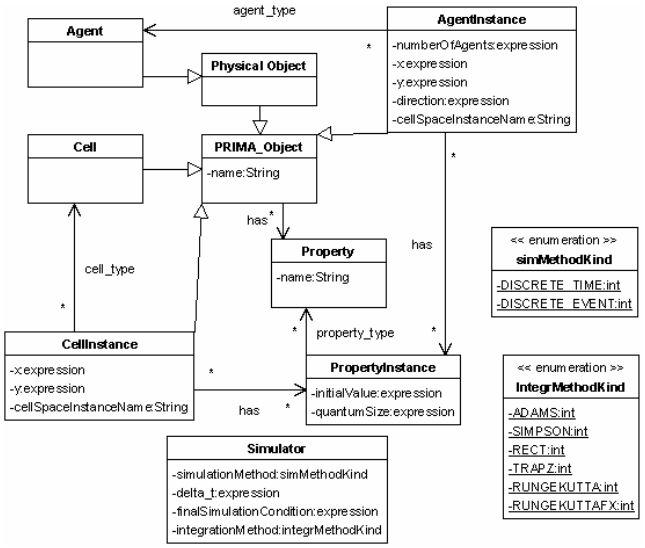

Figure 4: Instances Viewpoint.

4.4 Behavior

The *Behavior* viewpoint is used to specify agent or cell behaviors. Its metamodel is shown in Figure 5 and is similar to the UML Statecharts metamodel, but without nested states or orthogonal components.

The trigger of a transition can be either a call trigger (an event or a primitive function) or a time trigger. In the case of a primitive function trigger, this means that we can place here some of the methods of sensors (for example *receive()* of the communication sensor). Transitions are also provided with a guard, which may also contain references to primitive functions (for example, *seeObject()* of the vision sensor).

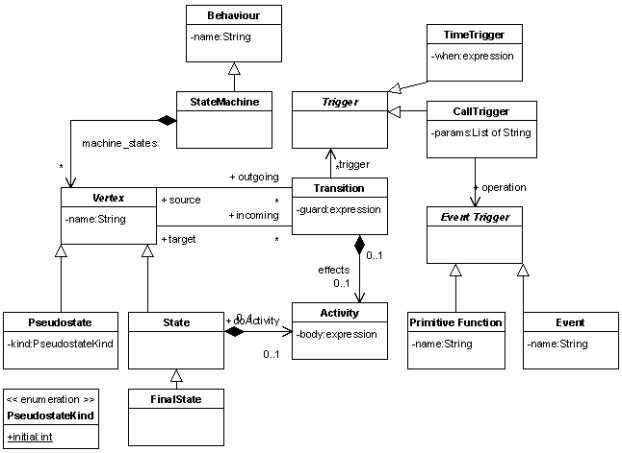

Figure 5: Behavior Viewpoint.

5. SPECIFICATION IN AToM^3

In order to describe a MVVL with AToM^3, the language designer starts building the metamodel for the complete VL. And then, he/she identifies the different viewpoints as restrictions of the complete VL. Thus, for each viewpoint, its name, attributes, metamodel and cardinality are specified. The cardinality is used in order to set the minimum and maximum numbers of instances of the viewpoint the final user can include in the project.

Figure 6 shows AToM^3 being used for the specification of the different viewpoints of PRIMA. The background window contains the complete metamodel of PRIMA. The tool to define the viewpoints is opened on top of this window, and shows the four viewpoints of PRIMA *(icons on the left)*: *Topology*, *Instances*, *Interactions* and *Behavior*. A viewpoint called *Repository (icon on the left)* is always present, and contains the complete MVVL metamodel. Viewpoint *Topology* is being edited in the figure. We have added three attributes (*modelName*, *authorName* and *description*) to describe the system view. A minimum cardinality of *1* has been specified, meaning that the user has to build at least one model of this type. When a viewpoint is created, the complete MVVL metamodel is copied, but the designer can delete elements, as the viewpoint should be a restriction of the complete MVVL metamodel.

Arrows between viewpoints are called consistency relations. There are two of them between each viewpoint and the repository. These are automatically generated by AToM^3 and contain TGGs to update the repository and to propagate changes to other views, as explained in the previous section. The VL designer can include additional consistency relations between specific viewpoints (with the corresponding TGG), but these are seldom necessary.

6. SIMULATING PRIMA MODELS

In PRIMA, Dynamic Structure Discrete Event Simulation (DSDEVS) simulation components [3] correspond to cells, agents and world. The latter embeds physical rules to coordinate interactions between agents and cells (atomic models), agents and physical objects, or agents and other agents. DSDEVS mechanisms are combined together to execute the high-level definitions of PRIMA.

An *Agent* component is a composite model composed of many atomic models. The atomic model *brain* pilots sensors and actuators to interact with the environment: the *comActuator* (for sending communication messages to other agents), the *propActuator* (to change properties), the *movActuator* (to describe the moving physical rules), the *comSensor* (to receive communication messages from other agents), the *visSensor* (to request visible objects to the World), and the *propSensor* (to request property values to the World). In addition, it is possible to have more than one sensor and actuator of each type. Each simulation component has a precise equivalence in the formal structure of DSDEVS models. For the case of sensors and actuators, we have built DEVS specifications which are the same independently of the model built by the user.

The brain DSDEVS specification is the most complicated, as it does not contain a predefined specification, but it is translated from the description of the agent behavior. The state space of the brain is made of an enumerate variable called *statechartState* containing one element for each state in the statechart. Another enumerate variable called *transitionAction* contains an element for each transition, plus an additional element "*none*". This variable is used to signal whether we are in the middle of executing an action associated with a transition. In the DSDEVS translation, we may have to break the execution of actions, as these may include the invocation of actuator methods. These method invocations to actuators are translated into DSDEVS as events passed from the brain component to the actuator component. Therefore, the action of the brain has to wait for the return of such method invocation. The state of the brain also has one variable for each property of the agent, and variables *X*, *Y*, *direction* and *id*. The latter is a unique agent identifier, useful for the simulation.

The external transition function of the brain takes care of receiving events from sensors and actuators, and has to check whether the brain is actually in the middle of the execution of a transition action. The output function sends events to actuators and sensors according to the transition actions of the behavior specification. The internal transition performs a change in state according to the statechart.

7. EXAMPLE

In this section we show the generated modeling environment and PRIMA is illustrated through its application to a bioeconomic model, simulated elsewhere [13]. The scope of the model is to determine under which biological and economic conditions an optimal management of the exploitation of a fish population leads to the creation of marine reserves. Simulation is used for the setting of virtual experiments. Agents are: Fishermen and a Manager. The latter is in charge of communicating authorizations to fishermen according to resources.

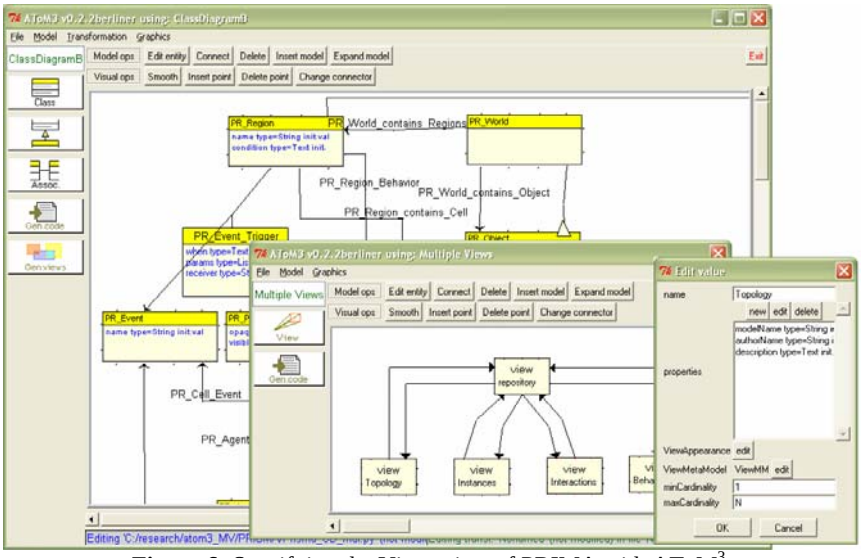

Figure 6: Specifying the Viewpoints of PRIMA with AToM^3.

From the previous definition of the PRIMA MVVL, a customized modeling environment was generated with the AToM^3 tool (see Figure 7). The environment allows the final user to create system views (in the background window) and checking their consistency by building a unique repository. This repository is automatically built by AToM^3 by executing the TGGs of the consistency relations. The user decides when to execute the consistency rules. She/he can see the repository, but cannot modify it. In particular, the figure shows that the user has created a topology view, an instances view, an interactions view and three behavior views. One of the latter views is being edited, showing two of the declared fields (author and model names), and the model. The model is a behavior automaton. We have defined the concrete syntax in such a way that connections from states to the state machine are transparent. Exactly one state machine object has to be present in each behavior system view. When the view is created, a state machine object is created. Moreover, we have also included an action in such a way that when the user creates a new state, it is connected to the state machine object automatically.

Each stock dynamics is distributed in space and represented by a system of differential equations. Each cell corresponds to one differential equation.

Figure 8 depicts the definition of two marine reserves represented as connected cellular spaces with a von Neumann neighborhood. Icons on the left allow to define: a *Cell Space Type*, a *Cell Space* instance (with name and dimensions). Connections are achieved through the upper *Connect* button. Source and target sides of cellular spaces can be connected using directions [*N*(orth), *S*(outh), *E*(ast), *W*(est)] and cells numbers.

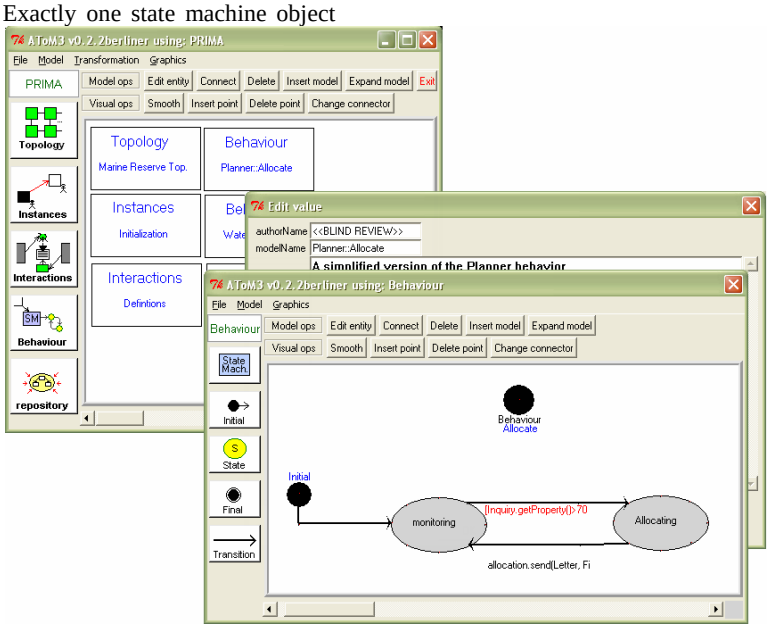

Figure 7: Generated Environment for PRIMA, Behavior System View Opened.

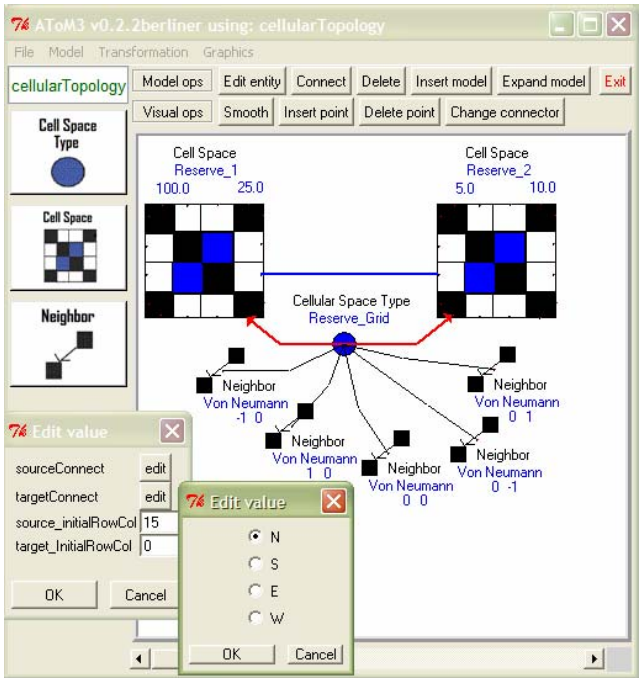

Figure 8: Marine Reserve Topology

Figure 9 describes interactions between agents in the environment. Icons on the left allow to define interacting types: *Cell*, *Agent*, *Behavior*, *Environment*, *Int Property*, *Property Sensor*, *Communication Sensor*, *Vision Sensor*, *Moving Actuator*, *Property Actuator*, *Communication Actuactor*, and *Communication Message*. Hence, fishermen have a moving actuator (a *Boat*), a property sensor (a *Radar*), a property actuator (a *Net*), and a communication sensor *Authorisation* for receiving a *letter* message from the fishery *Planner*. The latter performs inquiries about fish stocks to authorize fishermen or not, through a property sensor. The *Reserve Environment* embeds a *Fish Stock* property of each *Cell*. Behaviors (*Harvest*, *Evoluate*, and *Allocate*) are described through state automata. A very simplified version of the *Allocate* behavior is shown in Figure 7.

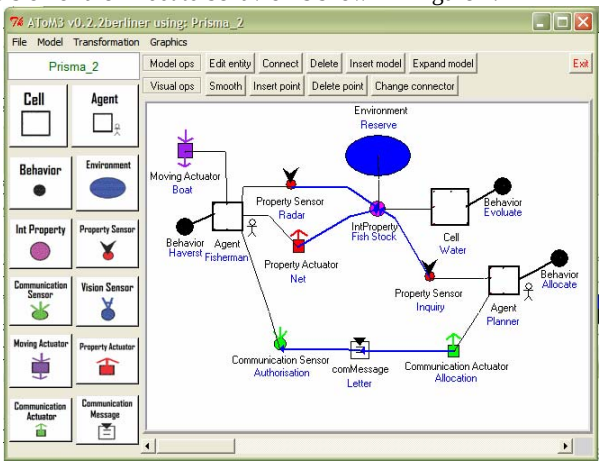

Figure 9: Marine Reserve Interactions.

Figure 10 depicts the different instances used (only one reserve is represented). Icons on the left allow to define: a *Cell Space Type* (of the same type of the *Cellular Topology* view), a *Cell Type*, an *Agent Type*, a *Property Type*, a *Cell Space* instance, a *Cell* instance, an *Agent* instance, a *Property* instance, and the initial conditions of the *Simulation*. Hence, *Reserve*_1 has an activity property for the detection of activity in a discrete time simulation [13]. *Fishermen* have a *Harvesting* property. *Cells* have a *Fish Stock*. The *Planner* (or *Local Administration*) has an *Implicit Price* [17] for each fisherman. According to this price and to the fish resources, the *Planner* can take the authorization decision. The simulation is a discrete time based. The Euler discretisation method is used.

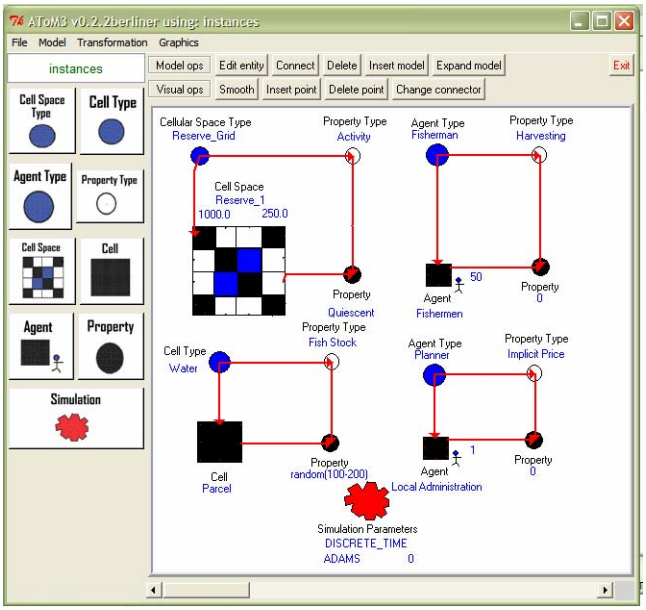

Figure 10: Marine Reserve Instances.

8. RELATED WORK

As stated in the introduction, there are many environments [1, 2, 19, 12] for agent-based simulation which offer frameworks for programming simulations. These approaches imply that the user should be proficient in the language the framework was coded in (Java in many cases). As agent-based models are frequently built by an interdisciplinary team, we believe this solution is not optimum. Even if line of codes can be used, visual models can be built and shown to communicate and develop models. PRIMA simplifies descriptions and facilitate the modeling phase for computer scientists. To achieve this goal, high-level descriptions and code generation are performed.

There have been many attempts to design modeling notations for designing Multi-Agent systems. For example, the FIPA modeling committee is working on AUML, an agent-based unified modeling language [14]. Although their aim is to be domain independent, we believe that in the area of simulation, many domain-specific concepts (sensors, actuators, cells, topologies, etc.) are necessary. Therefore, 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. The user is more productive using a customized modeling notation for agent-based simulation than using a more general one. In addition, the mapping into a formal language or code generation is more difficult in the case of a more generic language.

Finally, in [4], a more complex approach for the mapping from Statecharts to DEVS has been presented. In our case, due to our restricted Statecharts, a simpler algorithm can be used.

9. CONCLUSIONS AND FUTURE WORK

In this paper, PRIMA has been introduced, a multi-view visual language for agent-based modeling. The aim of the language is to provide high-level, visual constructs for building agent-based models. An environment for the language has been built using the metamodeling tool AToM^3 and its capabilities for defining multi-view languages.

The main perspective of this work is now to implement the translation of the PRIMA language into the formal simulation language DEVS [21]. Concurrency of processes will be tackled through parallel DEVS [21]. In parallel DEVS, simultaneous events can lead to the execution of one specific transition function (or action). Using this mechanism, the modeler should be free to specify when, *e.g.*, two agents are in the same cell (do they modify the same property at the same time?), if an event is generated by an agent and other agents have transitions enabled by that event, etc.

Furthermore, we are working in generating code for the VLE simulator [16]. Extensions of the language, for example for being able to dynamically create and destroy agents are also under consideration. For a validation of the language concepts and the mapping, we will apply the language to model industrial-scale simulations.

Acknowledgments

The authors acknowledge both anonymous reviewers who indicated precise issues to be discussed, mechanisms to be described and research directions to be explored.

REFERENCES

[1] Alfonseca, M., de Lara, J. *Two level evolution of foraging agent communities.* In BioSystems(66), Issues 1-2, 2002, pp.: 21-30. Elsevier.

[2] Ascape home page: http://www.brook.edu/es/dynamics/models/ascape/.

[3] Barros, F. J. 1995. *Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation*. Proc. Winter Simulation Conference, Arlington, pp.: 781-785.

[4] Borland, S., Vangheluwe, H. 2003. *Transforming Statecharts to DEVS*. Summer Computer Simulation Conference. Student Workshop, pages S154 - S159. Society for Computer Simulation International (SCS). Montréal, Canada.

[5] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer.

[6] France, R., Ghosh, S., Dinh-Trong, T., Solberg., A. 2006. *Model-Driven Development Using UML 2.0: Promises and Pitfalls*. In IEEE Software, February 2006, pp.: 59-66.

[7] Guerra, E., de Lara, J. 2006. *Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach*. Technical Report of the Universidad Carlos III (Madrid). Available at: www.ii.uam.es/~jlara/investigacion/techRep_UC3M.pdf

[8] Guerra, E., Díaz, P., de Lara, J. 2005. *Supporting the Automatic Generation of Advanced Modelling Environments with Graph Transformation Rules*. Proc. JISBD'05. pp.: 67-74. Thomson.

[9] Guerra, E., Díaz, P., de Lara, J. 2005. *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views*. In Proceedings of 2005 IEEE VL/HCC. Dallas. pp.: 284-286.

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

[11] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling*. Proc. ETAPS/FASE'02. LNCS 2306, pp.: 174 – 188. Springer. See also the AToM^3 home page: http://atom3.cs.mcgill.ca.

[12] 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.

[13] Muzy, A., Prunetti, D., and Innocenti, E. 2006. *A simulation-based tool for the creation of marine reserves.* IEEE International Symposium on Environment Identities and Mediterranean area (ISEIM 2006), Corte, Ajaccio, Corse, Accepted for publication.

[14] 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.

[15] Packard, N. H., Wolfram, S. 1985. *Two-Dimensional Cellular Automata*. Journal of Statistical Physics, 38 (March 1985) 901-946

[16] Ramat, E., Preux, P. 2003. *Virtual laboratory environment (VLE): a software environment oriented agent and object for modeling and simulation of complex systems*. Simulation Modelling Practice and Theory, Vol. 11, No. 1. pp. 45-55.

[17] Schaefer, M. B. 1954. *Some aspects of the dynamics of populations important to the management of commercial marine fisheries*. Bulletin of the Inter-American Tropical Tuna Commission 1: 25-56.

[18] Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars*. In LNCS 903, pp.: 151-163. Springer.

[19] Swarm home page: http://www.swarm.org.

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

[21] Zeigler, B. P., Praehofer, H., Kim T. G. 2000. *Theory of modelling and simulation*: Academic Press.