Agent based Modeling and Simulation with ActoMoS

Agostino Poggi
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parma, Italy
agostino.poggi@unipr.it

***Abstract* — ActoMoS is an actor-based software library for the development of agent-based models and for their simulation. This library offers software components and tools for modeling and simulating systems in different application domains. In particular, it allows the definition of agent model by reusing or extending a set of predefined agent models and supports efficient and scalable agent-based simulations involving a large number of agents. This paper, after an introduction of the actor model and implementation used by the software library, underlines the main features of the software library and presents its experimentation in some well-known domains.**

Keywords – Agent Based Modeling and Simulation, Actor model, Java.

I. INTRODUCTION

Simulation models are increasingly being used for solving problems and for helping in decision-making. The size and complexity of systems that are usually modeled are ever increasing. Modeling and simulation of such systems is challenging because it requires suitable and efficient modelling and simulation tools that take advantage of the power of current computing architectures, programming languages and software frameworks, and that make easy the development of applications.

Agent-based modeling and simulation (ABMS) tools and techniques seem be the most suitable means to cope with such challenges [19], [30]. In fact, ABMS has been and is widely used with success for studying complex and emergent phenomena in many research and application areas, including agriculture, biomedical analysis, ecology, engineering, sociology, market analysis, artificial life, social studies, and others fields. However, the limit of such tools and libraries is that their agent models shown a very limited use of the features offered by the computational agents found in Multi-Agent Systems (MAS) or Distributed Artificial Intelligence (DAI) techniques [12]. Therefore, it may be difficult to model some kinds of problem that, for example, require complex interaction among agents, and is usually less natural to distribute a simulation on a network of computational nodes.

This paper presents an actor based software library, ActoMoS, (Actor Modeling and Simulation) providing a set of suitable software components for the development of ABMS applications, the visualization of the simulations and the analysis of their results. The next section provides an overview of the software framework used for the implementation of the software library. Section 3 describes the features of the software library and shows how it makes easy the developing of agent based models and simulations. Section 4 shows its experimentation in some well-known ABMS application domains. Section 5 introduces related work. Finally, section 6 concludes the paper by discussing its main features and the directions for future work.

II. CODE SOFTWARE FRAMEWORK

CoDE (Concurrent Development Environment) is an actor-based software framework aimed at both simplifying the development of large and distributed complex systems and guarantying an efficient execution of applications [27]. CoDE is implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution. CoDE has a layered architecture composed of an application and a runtime layer. The application layer provides the software components that an application developer needs to extend or directly use for implementing the specific actors of an application. The runtime layer provides the software components that implement the CoDE middleware infrastructures to support the development of standalone and distributed applications.

In CoDE a system is based on a set of interacting actors that perform tasks concurrently. An actor is an autonomous concurrent object, which interacts with other actors by exchanging asynchronous messages. Communication between actors is buffered: incoming messages are stored in a mailbox until the actor is ready to process them. After its creation, an actor can change several times its behavior until it kills itself. Each behavior has the main duty of processing incoming messages through some handlers called cases. Each case can process only the messages that match a specific message pattern represented by an object that can apply a combination of constraints on the value of all the fields of a message. Therefore, if an unexpected message arrives, then the actor mailbox maintains it until a next behavior will be able to process it.

An actor can perform different types of action. It can send messages to other actors or to itself, create new actors, update its local state, set a timeout for waiting for the next message, change its behavior and kill itself. An actor can be viewed as a logical thread that implements an event loop [11], [21]. This event loop perpetually processes events representing the reception of messages, the exchange of behavior and the firing

of timeouts. In response on a reception of a message or the firing of a timeout, the actor finds and executes the suitable case for the processing of such an event. When the event represents the change of the behavior, the actor moves to the new behavior. In particular, the API of an actor does not offer any action for managing the reception of messages and for monitoring the firing of timeouts. In fact, an application developer uses an actor implementation provided by the CoDE runtime and needs only to provide the behaviors of the different actors of the application by defining the methods for their initialization and the message pattern – process method pairs of their cases.

Depending on the complexity of the application and on the availability of computing and communication resources, one or more actor spaces can manage the actors of the application and so an application can be distributed on a network of computational nodes. An actor space acts as "container" for a set of actors and provides the services necessary for their execution. In particular, an actor space takes advantages of two special actors: the scheduler and the service provider. The scheduler manages the concurrent execution of the actors of the actor space. The service provider enables the actors of an application to perform new kinds of action. The current implementation of the software framework provides services for supporting the broadcast of messages, the exchange of messages through the "publish and subscribe" pattern, the mapping of actors address to symbolic names, the mobility of actors, the interaction with users through emails and the creation of actors. The last service is important because an actor cannot directly create actors in other actor spaces, but can delegate it to their service providers.

One of the most important features of CoDE is the possibility of configuring an application with different implementations of the runtime components. It allows the use of different actor implementations, different schedulers and service providers. The type of the implementation of an actor is one of the factors that mainly influence the attributes of the execution of an application. In particular, actor implementation can be divided in two classes that allow to an actor either to have its own thread (from here named active actors) or to share a single thread with the other actors of the actor space (from here named passive actors). Moreover, the duties of a scheduler depend on the type of the actor implementation. Of course, a scheduler for passive actors is different from a scheduler for active actors, but for the same kind of actor can be useful to have different scheduler implementations. For example, it can allow the implementation of "cooperative" schedulers in which actors can cyclically perform tasks varying from the processing of the first message in the buffer to the processing of all the messages in it.

The most important decision that influence the quality of the execution of an application is the choice of the actor and scheduler implementations. In fact, the use of one or another couple of actor and scheduler causes large differences in the performance and in the scalability of the applications [6].

CoDE provides three types of actor implementation and four types of scheduler. The first two types of actor implementation represent active and passive actors. The third type of implementation represents special passive actors, called shared actors, which get messages from a shared queue. The first two types of scheduler implementation drive the execution of either active or passive actors (active and passive schedulers). The third type of implementation is used for shared actors (shared schedulers). Finally, the forth type of implementation is used in actor spaces containing both active and passive actors (hybrid schedulers).

The identification of the best couple of actor and scheduler implementations for a specific application mainly depends on the number of actors, the number of exchanged messages, the preeminent type of communication used by actors (i.e., point-to-point or broadcast) and the possible presence of a subset of actors that consume a large part of the computational resources of the application. Table 1 shows what should be the best choices for a qualitative partition of the values of the previous parameters. In particular, the third column indicates the preeminence of either point-to-point communication (P) or broadcast communication (B), the forth column indicates the presence/absence of a subset of heavy actors and the word "any" is used when the value of the associate parameter has not effect on the choice of actor and scheduler implementations.

TABLE 1

actors	**messages**	**P/B**	**Heavy**	**scheduler**
few	any	any	any	active
many	any	P	no	passive
many	few	B	no	passive
many	many	B	no	shared
many	any	any	yes	hybrid

Finally, an actor space can enable the execution of an additional runtime component called logger. The logger has the possibility to store (or to send to another application), in a textual or binary format, the relevant information about the execution of the actors of the actor space (e.g., creation and deletion of actors, exchange of messages, processing of messages and timeouts, exchange of behaviors and failures). Therefore, users and other applications can use such information for understanding the activities of an application, for diagnosing the causes of execution problems, and for solving them.

III. ACTOMOS

The features of the actor model and the flexibility of its implementation make CoDE suitable for building ABMS applications [28]. In particular, actors have the suitable features for defining agent models that can be used in ABMS applications and to model the computational agents found in MAS) and DAI systems. In fact, actors and computational agents share certain characteristics: i) both react to external stimuli (i.e., they are reactive), ii) both are self-contained, self-regulating, and self-directed, (i.e., they are autonomous), and iii) both interact through asynchronous messages and such messages are the basis for their coordination and cooperation (i.e., they are social). Moreover, given that actors interact only

through messages and there is not a shared state among them, it is not necessary to maintain an additional copy of the environment to guarantee that agents decide their actions with the same information (thing that is usually necessary in some application domain with other ABMS platforms). Finally, the use of messages for exchanging state information decouples the code of agents. In fact, agents do not need to access directly to the code of the other agents to get information about them, and so the modification of the code of a type of agent should cause lesser modifications in the code of the other types of agent. Finally, the use of actors simplifies the development of real computational agents in domain where, for example, they need to coordinate themselves or cooperate through direct interactions.

Moreover, the use of CoDE simplify the development of flexible and scalable ABMS applications. In fact, the use of active and passive actors allows the development of applications involving large number of actors, and the availability of different schedulers and the possibility of their specialization allow a correct and efficient scheduling of the agents in application domains that require different scheduling algorithms [20]. Moreover, the efficient implementation of broadcasting and multicast removes the overhead given to the need that agents must often diffuse the information about their state to the other agents of the application (e.g., their location in a spatial domain).

However, CoDE does not offer specific components for ABMS (e.g., simulators, agent models and simulation viewers). Therefore, we defined a software library, called ActoMoS (Actor Modelling and Simulation), that, starting from CoDE, provides a set of software components and tools for making easy the development of ABMS applications.

In large part of ABMS platforms usually a simulation is given by a sequence of steps where each agent needs only to get information about its surround (i.e., about a subset of the other agents and about the environment) and then to use such information for deciding its actions. In ActoMoS the simulation is similar, but agents get information about agents and the environment through messages.

In ActoMoS, to simplify the interaction between agents and the environment, the relevant parts of an environment are represented by a set of actors whose goals are to inform the agents acting in the environment about their presence and their state, and to update their state when the agents act on them. Given that the behavior of such actors is similar to the one expressed by the agents acting in the environment, we call both agents, but we divided them in active and passive agents. Active agents are the typical agents of an ABMS, i.e., they represent the entities able to move and cooperate with other entities acting in the environment. Passive agents define the environment of an ABMS, i.e., they represent the relevant elements of the environment (e.g., in a spatial domain the obstacles and the reference points for the movement of the active agents).

Such agents are usually implemented taking advantage of the shared actor implementation provided by CoDE, but it is necessary to develop a specific scheduler. Such a scheduler executes repeatedly all the agents and after each execution step broadcasts them a "clock" message. This last message allows to the agents to understand that they have all the information for deciding their actions, therefore, they decide, perform some actions and, at the end, broadcast the information about their new state.

In ActoMos, all the agents are usually represented by one or more actor behaviors that process the input messages through two cases. The first case processes the messages informing an agent about the state of the other agents. The second case processes the "clock" messages. However, while active agents exchange messages and perform other types of action (e.g., in a spatial domain to change their location), often, passive agents have the only duty of sending messages for informing the active agents about their presence (e.g., immutable obstacles or path points in a spatial domain). Therefore, such passive agents are represented by an actor behavior providing a case that get the "clock" messages for deciding when sending the information about their presence and state.

Of course, different types of agent have different implementations of the cases of their behaviors. In particular, ActoMoS provides some abstract behavior implementations for developing applications in different domains. Such implementations define the state information that an agent need to maintain in its specific application domain and provides a set of abstract methods for processing incoming information and for performing the actions in response to the "clock" messages.

Often the modelling of some systems (e.g., social networks) requires a massive number of agents. However, in such kind of systems, usually only a part of them is simultaneously active and the actions of the different agents do not need a synchronization. Therefore, it is necessary a scheduler that can manage a massive number of agents, but that can try to optimize the execution by scheduling only the active agents. The solution we implemented derives from the virtual memory techniques used by operating systems: agents increment an inactivity counter in the scheduling cycles in which they do not process messages and reset it in the cycles in which they process a message. The scheduler can get the value of such counters and can move an actor in a persistent store when its inactivity counter becomes greater than a fixed (or dynamic) threshold. The scheduler reload an actor from the persistent store when it receives a new message from another agent.

Of course, the number of active agents can vary over the simulation, but the quality of the simulation can be guaranteed if the number of the agents, maintained by the scheduler, remains in a range that depends on the available computational resources. The adopted solution, to limit to the number of active actors and to guarantee good performances, is to provide a scheduler able to move an inactive agent in the persistent storage on the basis of a variable number of inactive cycles. In particular, this number becomes high when the number of scheduled agents decreases (i.e., the scheduler does not spend time for storing agents in the persistence storage and reloading them) and becomes more and more low with the increasing of the number of scheduled agents.

Two important features that an ABMS framework should provide are the availability of graphical tools for the

visualization of the evolution of simulations and the possibility of analyzing the data obtained from simulations. CoDE does not provide any specific tool for ABMS, but provides a logging service that allows the recording of the Java objects describing the relevant actions of an actor (i.e., its initialization, reception, sending and processing of messages, creation of actors, change of behavior, and its shutdown). Therefore, we developed two graphical tools, that use such logging data for visualizing the evolution of simulations in spatial domains based on continuous and discrete 2D space representations, and another tool that use them for extracting statistical information about simulations. Fig. 9 shows two views of the GUI that supports 2D spatial simulations. In particular, it presents the initial and final views of the evacuation of a large number of pedestrians from a building.

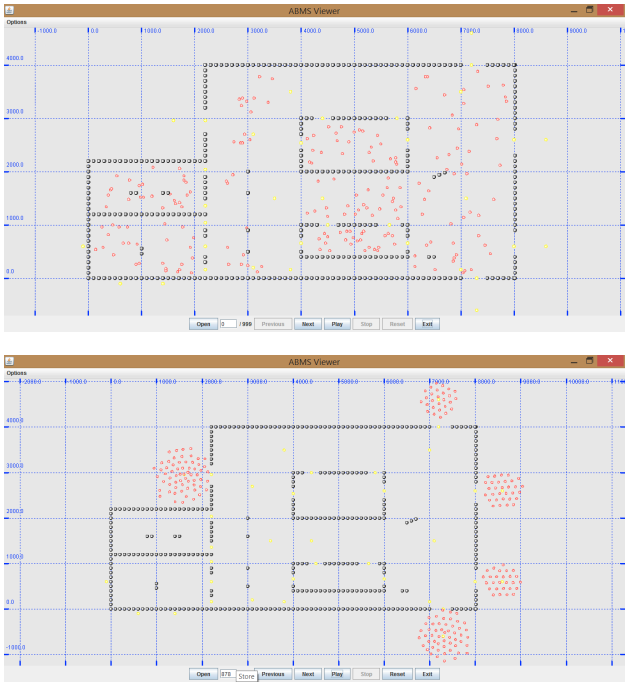

Fig. 9. Initial and final view of the simulation of a crowd evacuation.

IV. EXPERIMENTATION

We are using ActoMoS for the simulation of two well-known problems in a spatial environment: the prey-predator pursuit problem [2], and the crowd evacuation problem [33]. It is possible because ActoMoS offers all the software components necessary for modelling agents in a continuous or discrete 2D (or 3D) space (e.g., implementation of the algorithms that drive the movement of agents, agent models representing obstacles and path points).

The first experimentation of the prey-predator pursuit problem had be done in a 2D discrete space. Its main result is the definition of a flexible agent model that allows the execution of simulations with different algorithms, which drive the movement of the prey and of the predators, by simply changing the values of some configuration properties. Moreover, this agent model allows the solution of the conflicts among the moves of agents (i.e., two or more agents cannot share the same cell) with different coordination algorithms. The only constraint for using such coordination algorithms is that each agent needs to perform the current move, compute its next move and inform the other agents about it at each cycle. Therefore, each agent knows the intentions of the other agents before performing its move and performs the move only if the rules of the coordination algorithm allow it. This constraint does not cause a different behavior of the prey and the predators respect to the implementations of other ABMS platforms, because even in this implementation an agent can only either perform the previously decided move or remain in the same cell.

The first experimentation of the crowd evacuation problem had be done in a 2D continuous space and by implementing the agents of the crowd by using the boid model [31]. This experimentation take advantage of an extended set of boid rules that allow to agents to reach a meeting point outside the building by following either other agents or a set of alternative paths. Even in this case, the main result of the experimentation is the definition of a flexible agent model. Such a model allows the definition of different types of simulation by using different sets of boid rules. In particular, the experimentation shows how the possibility to follow other agents and the presence of paths towards the exit points make possible successful evacuations. Moreover, the use of the boid rules shows how is possible to obtain intelligent behaviors without using complex AI algorithms. However, the "calibration" of the model requires in some cases a large number of simulations to obtain a successful evacuation where agents both do not collide among them or with obstacles and do not lost time inside the building. In fact, the movement of each agent of the crowd is defined by the boid rules and so it is necessary to find the correct weights with which such rules contribute to the movement of the agent.

We are working for some time in the analysis and simulation of social networks [3], [4]. In particular, currently we are using ActoMoS for designing a peer-to-peer social network that can guarantee the same services of the centralized ones. The problem of peer-to-peer social network is that they do not have a centralized service that maintain the information shared among the users and so, for example, is difficult for a user that wakes up after a period of inactivity to get all the information of her/his interest that has been published when she/he was offline.

In particular, we defined a model of a peer-to-peer social network where a user can move from the online and offline state, publish new information and subscribe to new types of information. Each user is defined by a simple agent, which performs her/his actions using non-deterministic rules. Moreover, such an agent has also the duty to cooperate with the other agents to avoid that offline users lost information of their interest. Of course, the actors representing the online users should perform such a task. Moreover, its implementation should have the goals of limiting the amount of the recovered information (i.e., only the information of interest for the offline users should be recovered) and of guaranteeing privacy (i.e., users should not have additional information about the other users through the execution of such a task).

The experimentation is in the initial phase and we obtain good results with a simple algorithm of election that allows to the agents, representing the users that are moving offline, to assign the recording of the information of their interest, to agents of the users that remain online. The problems we need to solve is that in some situations there are few or no online users. In fact, when there are few users then their agents are overloaded in the recording of the information for the other users. When there are not users, when a user becomes online again it cannot have the lost information and cannot act as recorder for the other users. The solution that we are studying to solve such two problems is based on the introduction of "auxiliary agents", i.e., agents that: i) do not represent a user, ii) are running on computational nodes that are usually operative, and iii) have only the task of recording the information for the offline users.

V. Related Work

A lot of work has been done in the field of agent-based modeling and simulation. Moreover, some researchers used the actor model for the modeling and simulation of complex systems. The rest of the section presents some of the most interesting works presented in the previous two fields.

Swarm [22] is the ancestor of many of the current ABMS platforms. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents, and this paradigm is extended into the coding, including agent inspector actions as part of the set of agents. So in order to inspect one agent on the display, you must use another hidden, non-interacting agent. Swarm is a stable platform, and seems particularly suited to hierarchical models. Moreover, it supports good mechanisms for structure formation using multi-level feedback between agents, groups of agents, and the environment (all treated as agents).

Ascape [26] is a framework for developing and analyzing agent based models following some of the ideas of Swarm. However, it is somewhat easier to develop models with Ascape than with Swarm. Indeed, its goal is to allow people with only a little programming experience to develop quite complex simulations by providing a range of end user tools. Ascape is implemented in Java and users would require some ability to program in Java together with understanding of the object orientation philosophy.

NetLogo [32] is an ABMS platform based on the Logo programming language. Its initial goal was to provide a high-level platform allowing students, down to the elementary level, to build and learn from simple ABMS applications. Now it offers many sophisticated capabilities and tools that make it suitable for complex applications too. Moreover, a big advantage respect to the other platforms is the simplicity of its own language.

Repast [25] is a well-established ABMS platform with many advanced features. It started as a Java implementation of the Swarm toolkit, but rapidly expanded to provide a very full featured toolkit for ABMS. Although full use of the toolkit requires Java programming skills, the facilities of the last implementations allow the development of simple models with little programming experience [24].

MASON [16] is a Java ABMS tool designed to be flexible enough to be used for a wide range of simulations, but with a special emphasis on "swarm" simulations of a very many (up to millions of) agents. MASON is based on a fast, orthogonal, software library to which an experienced Java programmer can easily add features for developing and simulating models in specific domains.

ATC [7] is a framework for the modeling and validation of real-time concurrent systems based on the actor model. In particular, it inherits all the functional capabilities of actors and further allows the expression of most of the temporal constraints pertaining to real-time systems: exceptions, delays and emergencies.

The Adaptive Actor Architecture [17] is an actor-based software infrastructure designed to support the construction of large-scale multi-agent applications by exploiting distributed computing techniques for efficiently distribute agents across a distributed network of computers. This software infrastructure uses several optimizing techniques to address three fundamental problems related to agent communication between nodes: agent distribution, service agent discovery and message passing for mobile agents.

An actor-based infrastructure for distributing Repast models is proposed in [9]. This solution allows, with minimal changes, to address very large and reconfigurable models whose computational needs (in space and time) can be difficult to satisfy on a single machine. Novel in the approach is an exploitation of a lean actor infrastructure implemented in Java. In particular, actors bring to RePast agents migration, location-transparent naming, efficient communication, and a control-centric framework.

Statechart actors [10] are an implementation of the actor computational model that can be used for building a multi-agent architecture suitable for the distributed simulation of discrete event systems whose entities have a complex dynamic behavior. Complexity is dealt with by specifying the behavior of actors through "distilled" statecharts [14]. Distribution is supported by the theatre architecture [8]. This architecture allows the decomposition of a large system into sub-systems (theatres) each hosting a collection of application actors, allocated for execution on to a physical processor.

Simulator X [18] is a software research platform for intelligent interactive simulation that takes advantage of the actor model for supporting fine-grained concurrency and parallelism. The architecture uses actors to obtain a distributed application state and execution model. Simulator X is mainly used in the areas of real-time interactive systems, virtual reality and multimodal interaction.

VI. Conclusions

This paper presented a software library, called ActoMoS, which makes easy the development of agent-based models and supports efficient agent-based simulations involving a large number of agents. ActoMoS has been implemented on the top of CoDE (Concurrent Development Environment) that is an actor-based software framework aimed at both simplifying the

development of large and distributed complex systems and guarantying an efficient execution of applications [27].

ActoMoS has been experimented with success in the development of ABMS applications. Of course, its current implementation does not provide all the features of the most known ABMS platforms (i.e., NetLogo [32], Repast [25] and MASON [16]). However, the use of the actor model for the definition of agents allows to define real agent models where agent interact through the exchange of messages avoiding the use of a shared state and it simplifies the development of non-trivial applications where the management of concurrent activities may be of primary importance. Moreover, the availability of techniques to reduce the overhead of the diffusion of broadcast and multicast messages generally allows the development of applications whose performances are comparable with the ones provide by applications implemented by platforms that do not use messages for diffusing the state of the environment and of the agents of applications.

Current work has the goals of extending the functionalities of the software library and of continuing its current experimentation. Moreover, future work will be dedicated to the modeling and simulation of systems for e-business services [23], collaborative work services [13] and for the management of information in pervasive environment [5][29].

REFERENCES

[1] G.A. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems," Cambridge, MA, USA: MIT Press, 1986.

[2] M. Benda, V. Jagannathan, and R. Dodhiawalla, "On optimal cooperation of knowledge sources," Tech. Rep. BCS-G2010-28, Boeing AI Center, Bellevue, WA, USA, 1986.

[3] F. Bergenti, E. Franchi, and A. Poggi,. "Selected models for agent-based simulation of social networks," in Proc. of 3rd Symp. on Social Networks and Multiagent Systems (SNAMAS 2011), York, UK. 2011, pp. 27-32.

[4] F. Bergenti, E. Franchi and A. Poggi, "Agent-based interpretations of classic network models," Computational and Mathematical Organization Theory, Vol. 19, No. 2, 2013, pp. 105-127, 2013.

[5] F. Bergenti, and A. Poggi, "Ubiquitous Information Agents," International Journal on Cooperative Information Systems, Vol. 11, No. 3-4, pp. 231-244, 2002.

[6] F. Bergenti, A. Poggi, and M. Tomaiuolo, "An Actor Based Software Framework for Scalable Applications," in Internet and Distributed Computing Systems, Berlin, Germany: Springer, 2014, pp. 26-35.

[7] L. Boualem, and S. Yamina, "On equivalences for actors expressions and configurations of ATC," WSEAS Transactions on Computers vol. 4, no. 9, pp. 1045-1053, 2005.

[8] F. Cicirelli, A. Furfaro, and L. Nigro, "Exploiting agents for modelling and simulation of coverage control protocols in large sensor networks," Journal of Systems and Software, vol. 80, no. 11, pp. 1817-1832, 2007.

[9] F. Cicirelli, A. Furfaro, A. Giordano and L. Nigro, "Distributing Repast simulations using actors," in Proc. of 23rd European Conf. on Modelling and Simulation, Madrid, Spain, 2009, pp. 226–231.

[10] F. Cicirelli, A. Furfaro, and L. Nigro, "Modeling and simulation of complex manufacturing systems using Statechart-based actors," Simulation Modelling Practice and Theory, vol. 19, no. 2, pp. 685–703, 2011.

[11] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt and W. De Meuter, "Ambient-oriented programming in ambienttalk," in ECOOP 2006 – Object-Oriented Programming, Berlin, Germany: Springer, 2006, pp. 230-254.

[12] A. Drogoul, D. Vanbergue, and T. Meurisse, "Multi-agent based simulation: Where are the agents?," In Multi-agent-based simulation II, Berlin, Germany: Springer, 2003, pp. 1-15.

[13] E. Franchi, A. Poggi, and M. Tomaiuolo. "Open social networking for online collaboration," International Journal of e-Collaboration, vol. 9, no. 3, pp. 50-68, 2013.

[14] D. Harel, and A. Naamad, "The Statemate semantics of Statecharts," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 4, pp. 293–333, 1996.

[15] C. E. Hewitt, "Viewing controll structures as patterns of passing messages," Artificial Intelligence, vol. 8, no. 3, pp. 323–364, 1977.

[16] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A multiagent simulation environment," Simulation, vol. 81, no. 7, pp. 517-527, 2005.

[17] M. Jang, and G.A. Agha, "Scalable agent distribution mechanisms for large-scale UAV simulations," in Proc. of Int. Conf. of Integration of Knowledge Intensive Multi-Agent Systems, Waltham, MA, USA, 2005.

[18] M. E. Latoschik, and H. Tramberend. "A scala-based actor-entity architecture for intelligent interactive simulations," Proc. of 5th IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2012, pp. 9-17.

[19] C.M. Macal, and M.J. North, "Tutorial on agent-based modelling and simulation, Journal of Simulation," vol. 4, no. 3, pp. 151–162, 2010.

[20] P. Mathieu, and Y Secq, "Environment Updating and Agent Scheduling Policies in Agent-based Simulators," in Proc. of 4th Int. Conf. on Agents and Artificial Intelligence (ICAART), Algarve, Portugal, 2012, pp. 170-175.

[21] M. S. Miller, E. D. Tribble, and J. Shapiro, "Concurrency among strangers," in Trustworthy Global Computing, Berlin, Germany: Springer, 2005, pp. 195-229.

[22] N. Minar, R. Burckhart, C. Langton, and V. Askenasi, 1996. "The Swarm simulation system: a toolkit for building multi-agent systems," Santa Fe Institute, Santa Fe, NM, USA. http://www.swarm.org/ [Accessed March 25, 2015].

[23] A. Negri, A. Poggi, M. Tomaiuolo, and P. Turci, "Agents for e-Business Applications," in 5th Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Hakodate, Japan: ACM., 2006, pp. 907-914.

[24] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos, "The Repast Simphony runtime system," in Proc. of the Agent 2005 Conf. on Generative Social Processes, Models, and Mechanisms, Chicago, IL, USA, 2005

[25] M. J. North, N. Collier, and J. Vos, "Experiences in creating three implementations of the repast agent modeling toolkit," ACM Transactions on Modeling and Computer Simulation, vol. 16, no. 1, pp. 1-25, 2006.

[26] M. T. Parker, "What is Ascape and why should you care," Journal of Artificial Societies and Social Simulation, vol. 4, no. 1, 2001.

[27] A. Poggi, "Developing Scalable Applications with Actors," WSEAS Transactions on Computers vol. 14, pp. 660-669, 2014.

[28] A. Poggi, "Replaceable Implementations for Agent-Based Simulations," SCS M&S Magazine, vol. 4, no. 3, pp. 1-10, 2014.

[29] A. Poggi, and M. Tomaiuolo, "A DHT-based multi-agent system for semantic information sharing," in New Challenges in Distributed Information Filtering and Retrieval, Berlin, Germany: Springer, 2013, pp. 197-213.

[30] S.F. Railsback, S.L. Lytinen, and S.K. Jackson, "Agent-based simulation platforms: Review and development recommendations," Simulation, vol. 82, no. 9, pp. 609–623, 2006.

[31] C.W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," ACM SIGGRAPH Computer Graphics, vol. 21, no. 4, pp. 25-34, 1987.

[32] S. Tisue, and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in Proc. of Int. Conf. on Complex Systems (ICCS 2004), 16-21, Boston, MA, USA, 2004, pp. 16-21.

[33] M.H. Zaharia, F. Leon, F. C. Pal, and G. Pagu, Agent-based simulation of crowd evacuation behavior, in Proc. of 11th WSEAS Int. Conf. on Automatic control, modelling and simulation (ACMOS'09), Istanbul, Turkey, 2009, pp. 529-533.