An Actor-Based Software Framework for Developing and Simulating Complex Systems

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

***Abstract* —ASiDE is an actor-based software framework that has the goals of simplifying the development of large and distributed complex systems and of guarantying an efficient execution of applications. This software framework provides a flexible actor implementation that simplifies the writing of the actors by delegating the management of events (i.e., the reception of messages) to the execution environment, and allowing the choice between an active thread solution (i.e., each actor has its own thread) and a passive thread solution (i.e., several actors share the same thread). In particular, the second thread solution is suitable to implement systems whose behavior should be modeled through the use of a large number of actors. ASiDE is currently used for proving its advantages for the development of agent based modeling and simulation tools.**

Keywords - Actor model, software framework, concurrent programming, distributed systems, ABMS.

I. Introduction

Computing architectures are getting increasingly distributed, from multiple cores in one processor and multiple processors in a computing node, to many computing nodes. It demands a bigger need for distributed and concurrent programming because sequential programming models are not suitable to such kinds of architectures.

Distributed and concurrent programming is hard and is not like that of sequential programming. Programmers have more concerns when it comes to taming parallelism. Distributed and concurrent programs are usually bigger than equivalent sequential ones. Models of distributed and concurrent programming languages are different from familiar and popular sequential languages [11][18].

Message passing models seems be the more appropriate given that they can cope with all the problems caused from the sharing of data among the concurrent parts of a system. One of the well-known theoretical and practical models of message passing is the actor model. Using such a model, programs become collections of independent active objects (actors) that exchange messages and have no mutable shared state [1][2][9]. Actors can help developers to avoid issues such as deadlock, live-lock and starvation, which are common problems for shared memory based approaches. There are a multitude of actor oriented libraries and languages, each of which implements some variant of actor semantics. However, such libraries and languages had to choose between to make simple the writing of the code (by using the thread based programming model) and to allow the development of efficient large systems (by using the event based programming model).

Simulation models are increasingly being used to solve problems and to aid in decision-making. The size and complexity of systems which are usually modeled (e.g., communication networks, biological systems, weather forecasting, manufacturing systems, etc.) is ever increasing. Modeling and simulation of such systems is challenging in that it requires suitable and efficient simulation tools that take advantage of the power of current computing architectures and, of course, of programming languages and software frameworks that can exploit such kinds of architecture and that offer the features useful for the development of such kinds of system. In particular, agent-based modeling and simulation (ABMS) tools and techniques seem be the most suitable means to exploit the power of such computing architectures [13][19].

This paper presents an actor based software framework, called ASiDE (Actor based Simulation and Development Environment), that has the features for simplifying the development of large and distributed complex systems and for guarantying an efficient execution of applications. The next section introduces related work. Section three describes the software models that are the basis of the framework. Section four describes the features of its current implementation. Section five introduces its initial experimentation for agent based modeling and simulation. Finally, section six concludes the paper by discussing its main features and the directions for future work.

II. Related Work

Several actor oriented libraries and languages have been proposed in last decades and a large part of them use Java as implementation language. A lot of work has also been done in the field of agent-based modeling and simulation. Moreover, some works used the actor model for the modeling and simulation of complex system. The rest of the section presents some of the most interesting works presented in the previous three fields.

Salsa [22] is an actor-based language for mobile and Internet computing that provides three significant mechanisms based on actor model: token-passing continuations, join continuations, and first-class continuations. In Salsa each actor has its own thread, and so scalability is limited. Moreover, message passing performance suffers from the overhead of reflective method calls.

Kilim [21] is a framework used to create robust and massively concurrent actor systems in Java. It takes advantage of code annotations and of a byte-code post-processor to simplify the writing of the code. However, it provides only a very simplified implementation of an actor model where each actor (called task in Kilim) has a mailbox and a method defining its behavior. Moreover, it does not provide remote messaging capabilities.

Scala [7] is an object-oriented and functional programming language that provides an implementation of the actor model unifying thread based and event based programming models. In fact, in Scala an actor can suspend with a full thread stack (receive) or it can suspend with just a continuation closure (react). Therefore, scalability can be obtained by sacrificing code writing simplicity. In particular, Scala has been used for the development of Scalation [14]. Scalation is a domain specific language which supports multi-paradigm simulation modeling, including dynamics, activity, event, process and state oriented models.

Jetlang [20] provides a high performance Java threading library that should be used for message based concurrency. The library is designed specifically for high performance in-memory messaging and does not provide remote messaging capabilities.

Swarm is the ancestor of many of the current ABMS platforms [15]. 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 through the use of multi-level feedback between agents, groups of agents, and the environment (all treated as agents).

Repast is a well-established ABMS platform with many advanced features [17]. 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 [16].

MASON 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 [12]. 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.

The Adaptive Actor Architecture [10] is an actor-based software infrastructure designed to support the construction of large-scale multi-agent applications by exploiting distributed

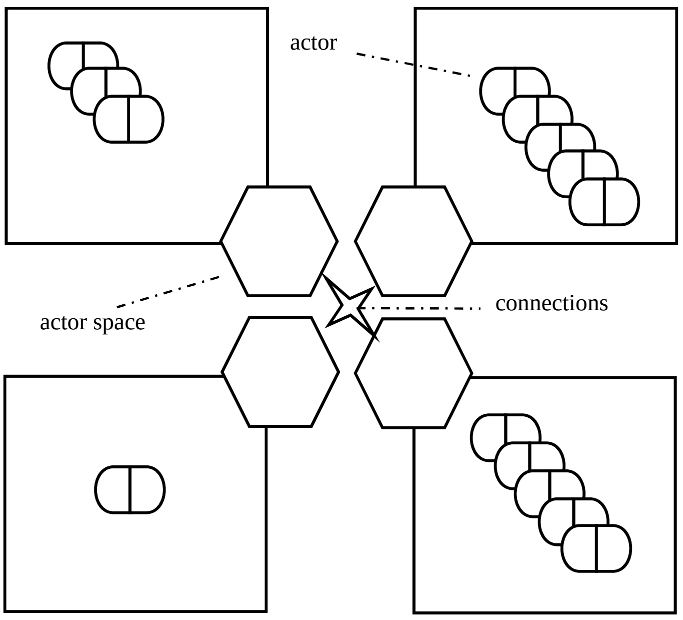

Figure 1. ASiDE system architecture.

computing techniques to 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 [17] is proposed in [4]. 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 [5] 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 [8]. Distribution is supported by the theatre architecture [3]. 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.

III. ASiDE

ASiDE (Actor based Simulation and Development Environment), is an actor based software framework that has the goals of simplifying the development of large and distributed complex systems and of guarantying an efficient execution of applications. This software framework provides a flexible actor implementation that simplifies the writing of the actors by allowing the choice between an active thread solution (i.e., each actor has its own thread) and a passive thread solution (i.e., several actors share the same thread), and by delegating the management of events (i.e., the reception of messages) to the execution environment on the basis of the current thread solution (i.e., actors execution is simply blocked for waiting for messages when the active thread solution is used, actors execution is scheduled only when they receive new messages when the passive thread solution is used.

A. Actors

In ASiDE 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.

Each actor has a unique mail address which is used to specify a target for communication. After its creation, an actor can change several times its behavior until it kills itself. Each behavior has the main duty of processing a set of specific messages. Therefore, if an unexpected message arrives, then it is maintained in the actor mailbox until a next behavior will be able to process it.

An actor can perform five types of action:

- It can send messages to other actors or to itself.
- It can create new actors.
- It can update its local state.
- It can change its behavior.
- It can kill itself.

An actor has not direct access to the local state of another actor and can exchange data with another actor only when it creates a new actor or when it sends a message. However, it is its implementation that will (or will not) guarantee that actor creation and message passing actions do not allow the sharing of mutable data.

An actor can send messages only to the actors of which it knows the address, that is, the actors it created and the actors of which it received their addresses through a message. An actor can send messages that require or not an answer and that are replies to the messages of other actors. In particular, an actor can perform four different actions for sending messages:

- It can send a message to another actor without requiring an answer.
- It can send a message to another actor requiring an answer.
- It can reply to a message of another actor without requiring an answer.
- It can reply to a message of another actor requiring an answer.

An actor has not explicit actions for the reception of messages, but its implementation will autonomously manage the reception of messages and then will execute the actions for their processing.

An actor has the possibility of setting and then modifying a timeout within its current behavior must receive a message. However, it has not explicit actions for managing the firing of such a timeout: its implementation will autonomously observe the firing of the timeout and then will execute the actions for its management.

B. Actor Spaces

Depending on the complexity of the system and on the availability of computing and communication resources, actors can be aggregated in one or more actor spaces. Figure 1 shows a graphical representation of the architecture of an ASiDE system.

An actor space is a concurrent object that acts as container of a set of actors. In particular, an actor space supports a transparent communication between local actors and remote actors (i.e., the actors of the other actor spaces) and enhances their functionalities through a set of services (e.g., the broadcasting of messages to local actors and the migration of local actors to other actor spaces).

Actors can require the execution of such services by sending a message to an actor space. Therefore, even an actor space has a unique name (address) and a mailbox. Moreover, local actors know a priori the address of their actor space and remote actors can receive the address of the other actor spaces of a system through some messages.

IV. Implementation

ASiDE is implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution.

The architecture of an ASiDE application can be divided in an application and a runtime layer. The application layer is represented by the specific actors and actor spaces of the application. The runtime layer implements the ASiDE middleware infrastructures to support the development of applications based on several actors distributed on a set of different actor spaces.

The current implementation, besides building the components of the runtime layer, provides some components (i.e., interfaces, concrete and abstract classes) to simplify the development of the actors of an application.

The design of the software has been planned with the goal of guaranteeing an easy development of several implementations of the components of the runtime layer that take advantages of different technologies and algorithms without the need of modifications to the application layer (i.e., the code of the actors). This feature is important for enabling the use of different solutions for the management of the execution of the actors running inside the same actor space, but also for supporting the communication among actors of different actor spaces.

An actor is composed of two main parts: a plugger and a behavior. A plugger is a component of the runtime layer that provides the main functionalities of an actor (i.e., the threading solution, the way in which it is executed and the implementation of its actions). A behavior is a component of the application layer that defines the actions that an actor will perform during a specific part of its life. Therefore, the

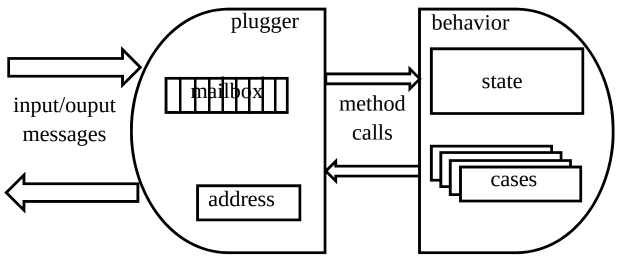

Figure 2. Actor architecture

development of an application involves, besides the deployment of the actors in different actor spaces, the configuration of the pluggers used in the application and the development of the code of the behaviors used by the actors of the application. Figure 2 shows a graphical representation of the architecture of an actor.

In ASiDE, an actor is defined by extending the *Actor* abstract class and by implementing the methods driving its behavior. In particular, this kind of implementation implies that when an actor moves from a behavior to another one, an actor object is replaced by another one. Of course, the sequence of actor objects that drive the life of an actor share the same address and mailbox objects. To do it, ASiDE implements the creation of actors through a factory that creates a plugger (containing the address and the mailbox of the actor) and passes it to the actor constructor, and implements the change of behavior through the actor plugger that passes its reference to the constructor of the next actor.

TABLE 1

Field name	**Field description**
Identifier	Identifier oif the message
Sender	Address of the sender actor
Receiver	Address of the receiver actor
Content	Content of the message
Time	Delivery time of the message
NeededReply	True if the message needs a reply
InReplyTo	Identifier of the replied message

The original actor model associates a behavior with the task of messages processing. In ASiDE, a behavior can perform three kinds of tasks: an initialization task, the management of message reception timeouts, and the processing of messages. Initialization is performed by the *initialize* method and timeouts management is performed by the *getTimeout*, *setTimeout* and *timeout* methods. In particular, the *initialize* and *timeout* methods can be overridden for defining specific behaviors.

The processing of messages is performed by some case objects. A case has the goal of processing the messages that match a specific (and fixed) message pattern. In ASiDE, a case is defined by extending the *ActorCase* abstract class and by implementing its *process* method. The code of the methods that drive the behavior of an actor (i.e., *initialize*, *timeout* and *process*) is built on a set of methods that implement the actions for sending messages, for creating other actors, for changing the behavior and for killing itself. Of course, the updating of the local state can obtained through the use of either the simple operators of the Java language or through the methods implementing the actor actions.

A plugger acts as interface between an actor and the runtime layer and provides the implementation of the actions that an actor can perform. ASiDE provides different configurable implementations of a plugger. Therefore, the use of the possible different implementations and configurations of a plugger allow providing different behaviors and performances for the same application. In particular, the current plugger implementations and configurations allow: i) to have two different thread solutions (i.e., actor of an actor space either have their own thread or share a single thread), ii) to enable the logging of the main actor activities (e.g., the sending of messages and the processing of messages and timeouts), iii) to impose the exchange of only immutable objects between actors, and iv) to use different scheduling algorithms in the case that actors share a single thread.

A message is an object that contains a set of fields maintaining the typical header information and a field maintaining the exchanged data. TABLE 1 introduces a short description of the field of a message.

TABLE 2

Constraint name	**Constraint description**
All(c, O)	True if all the objects of a collection satisfy the constraint
And(C, o)	True if the object satisfies all the constraints
IsDifferent(o1, o2)	True if the object is different from the reference object
IsEqual(o1, o2)	True if the object is equal to the reference object
IsHigher(o1, o2)	True if the object is greater than the reference object
IsInstance(t, o)	True if the object is an instance of the reference type
IsLower(o1, o2)	True if the object is less than the reference object
IsNull(o)	True if the object is null
IsOneOf(O, o)	True if the object is one of the set of objects
Matches(p, o)	True if the object matches the pattern
Not(c, o)	True if the object does not satisfy the constraint
Or(C, o)	True if the object satisfies at least one of the constraints
Some(c, O)	True if at least one of the objects of a collection satisfies the constraint

As introduced above, an actor processes the received messages through a set of case objects and each of them can process only the messages that match a specific message pattern. In ASiDE, a message pattern is an object of the *MessagePattern* class that can apply constraints on the value of all the fields of a message. A constraint is also an object of one of a predefined set of classes that implement the *Constraint* interface. TABLE 2 introduces a short description of the possible constraints.

This kind of representation of a message pattern allows a very sophisticate filtering of messages. Moreover, the use of

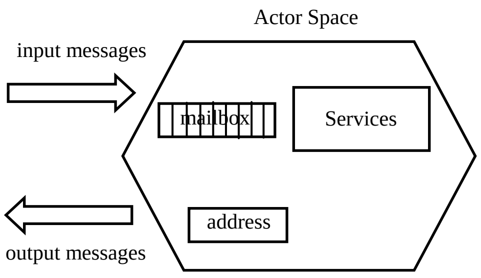

Figure 3. Actor space architecture

the *Matches* constraint allows a specialization of the filtering for all the message fields and in particular for the content of the message. The current implementation of ASiDE provides an additional pattern, implemented by the *RegularExpression* class. This pattern allows the filtering of the messages by matching the string representation of the value of a field with a specific regular expression.

New types of pattern and constraint can be added. Of course, it requires working at the application and runtime layers. At the application layer, it is necessary to develop the classes representing such patterns and/or constraints. At the runtime layer, it is necessary to develop the software components able to uses such patterns and/or constraints for filtering messages.

An actor space is a component of the runtime layer that, as introduced, above, has the duty of both supporting the communication between local actors and remote actors and to provides them a set of services. Figure 3 shows a graphical representation of the architecture of an actor space.

The implementation of the actor space allows the deployment of a system with different communication technologies and with different set of services. However, while all the actor spaces of a system need to use the same communication technology; anyone of them can provide a different set of the services. In fact, services are divided in mandatory and optional, and an actor space must provide all the mandatory services and can choose what of the others to provide. New kinds of service can be added. Of course, it requires working at the application and runtime layers. At the application layer, it is necessary to develop the classes representing the service requests that actors need to send to the actor spaces. At the runtime layer, it is necessary to develop the software components able to perform the services.

The current implementation of ASiDE allows building both standalone and distributed applications and associated an actor space with a Java virtual machine. A standalone application is based on a single actor space created with an initial agent that will start the application execution. A distributed application is based on more than one actor space. Each actor space can have an initial agent, but usually a distributed application start with a set of empty actor spaces and an "initiator" actor space with an initial actor that also starts one or more actors on the other actor spaces.

The performances of an application, its behavior and the possibility of monitoring and debugging its execution depend on its configuration. In fact, the configuration step allows the choice of:

- The threading solution for each actor space: active, actors have their own thread, or passive, actors share a single thread.
- The algorithm used for scheduling the actors of each actor space (in the case of passive threading solution).
- The services provided by each actor space.
- The communication technology used for connecting the actor spaces of the application.
- The activities of the actor and actor spaces that must be logged.

The choice that mainly determinates the performances of an application is the threading solution. In fact, while with the active threading solution each actor of an actor space is executed in a distinct Java thread, with the passive thread solution all the actors of an actor space share a single Java thread and their execution is managed through a scheduler. The choice of the threading solution may influence the behavior of the application. In fact, using the active threading solution, different executions of the application can have different behaviors because of the different arrival order of messages (of course, if the application is started with the same initialization information). Using the passive threading solution, the simplest scheduling algorithms guarantee the same arrival order of messages for different execution of the application.

V. USING ASiDE FOR ABMS

The features of the actor model and the flexibility of its implementation make ASiDE suitable for building agent based modeling and simulation tools. In fact, the use of passive actors allows the development of applications involving large number of actors, and the possibility of using different schedulers for their execution allows the development of schedulers that are specialized for some specific application domains.

The first experimentation of ASiDE for modeling and simulating systems has been dedicated to the well-known game of life [6]. The model of such a game is based on a grid of *LifeCell* actors that have been created by a *Creator* actor and initialized by a set of Initializer actors that send them the acquaintance (i.e., the address) of the neighbors actors The behavior of the *LifeCell* actor is very simple given that it cycles on the following three tasks:

- Get messages from neighbors.
- Compute the new state.
- Send messages to neighbors.

The implementation of the actor involves the implementation of the *initialize* and *timeout* methods and the creation of a case for processing of the messages coming from the neighbors. In particular, while the *initialize* method sets the initial state of the cell, the *timeout* method computes the new state and then send a message to the neighbor. Finally, the *process* method of the case of the actor needs only to count the messages notifying that a neighbor is living.

This experimentation gave the opportunity to make some measures of the performance of the ASiDE software for developing simulations involving different numbers of actors. TABLE 3 presents the execution times of two simulations of the “game of life” having a length of respectively a hundred and a thousand of cycles and involving from few hundreds to more than a million of actors. These results were obtained on a laptop with an Intel Core 2 - 2.80GHz processor, 8 GB RAM, Windows 7 OS and Java 7 with 4 GB heap size. Moreover, all the performed experimentations showed that the use of the passive threading solution provides better performances than the active one even with a few number of actors.

TABLE 3

Actors number	**Execution time (ns)**	
	100 cycles	**1000 cycles**
256	176.943.577	260.195.988
1.024	233.455.065	419.241.045
4.096	605.946.575	1.995.799.263
16.384	3.420.273.304	20.098.356.918
65.536	14.203.557.943	90.057.744.911
262.144	57.711.405.356	372.067.196.751
1.048.576	222.013.993.687	1.375.419.844.501

VI. CONCLUSIONS

This paper presented a software framework, called ASiDE, which uses the most natural programming model for writing actors code (i.e., the thread based model), but allows also the development of efficient large systems by combining the sharing of threads among the actors of the systems with the delegation of the management of the reception of messages to the execution environment.

ASiDE is implemented by using the Java language and its use can simplify the development of systems in heterogeneous environments where computers, mobile and sensor devices must cooperate for the execution of tasks.

We are using the ASiDE software framework for the development of some applications in the fields of distributed information sharing, social networks and ABMS. A first analysis of the experimentation shows that each computing node (actor space) can efficiently run more than a million of actors and the work of a system can be distributed on several computational nodes with a limited overhead in a local network.

Future research activities will be dedicated, besides to continue the experimentation and validation of the software framework in the distributed information sharing and for using ABMS in the field of social networks modeling and analysis, and to the improvement and the extension of the software framework. In particular, current activities are dedicated to: i) allow the mobility of actors between different actor spaces, ii) provide a passive threading solution that fully take advantage of the feature of multi-core processors, iii) to define a set of actors schedulers and supporting tools for performing simulations and the analysis of the results of the simulations in some specific application domains, iv) to improve the scalability of simulations on the number of actors through the definition of distributed actors schedulers, and v) provide a set of actor space services to simplify the coordination among the actors of a system (e.g., group communication protocols) and to enable the interoperability with Web services and legacy systems.

REFERENCES

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

[2] G.A. Agha, I.A. Mason, S.F. Smith, and C.L. Talcott, A Foundation for Actor Computation. Journal of Functional Programming, 7(1):1-69, 1997.

[3] F.Cicirelli, A. Furfaro, and L. Nigro. “Exploiting agents for modelling and simulation of coverage control protocols in large sensor networks”. The Journal of Systems and Software, 80(11):1817-1832, 2007.

[4] F.Cincirelli, Distributing ResPast simulations using actors. in: Proc. of 23rd European Conference on Modelling and Simulation (ECMS’09), 9–12 June, Madrid, 2009, pp. 226–231.

[5] F. Cincirelli, A. Furfaro and L. Nigro. Modeling and Simulation of Complex Manufacturing Systems using Statechart-based Actors. Simulation Modelling Practice and Theory, Volume 19, Issue 2, Pages 685–703, 2011.

[6] M. Gardner, The fantastic combinations of John Conway's new solitaire game Life. Scientific American 223:120-123, 1970.

[7] P. Haller and M. Odersky, Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science, 410(2-3):202–220, 2009.

[8] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. ACM Trans. on Soft. Eng. And Methodology (TOSEM), 5(4): 293–333. 1996.

[9] C.E. Hewitt, Viewing controll structures as patterns of passing messages. Artificial Intelligence, 8(3):323–364, 1977.

[10] M. Jang, and Gul Agha, "Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulations," Proc. Int. Conf. of Integration of Knowledge Intensive Multi-Agent Systems, Waltham, MA, 2005.

[11] C. Leopold, Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches. John Wiley & Sons, Inc., New York, NY, 2001.

[12] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, MASON: A Multiagent Simulation Environment. Simulation 81, 7, 2005, 517-527.

[13] C.M. Macal and M.J. North, Tutorial on agent-based modelling and simulation, Journal of Simulation (2010) 4, 151–162.

[14] J.A. Miller, J. Han, and M. Hybinette, “Using Domain Specific Language for Modeling and Simulation: ScalaTion as a Case Study," In Proc. of the 2010 Winter Simulation Conference, pp.741-752, 2010.

[15] N. Minar, R. Burckhart, C. Langton, and V. Askenasi, The Swarm Simulation System: a Toolkit for Building Multi-Agent Systems. 1996. Available from http://www.swarm.org/

[16] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos. The repast simphony runtime system. In Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms, 2005.

[17] M. North, N. Collier, and J. Vos. Experiences in creating three implementations of the repast agent modeling toolkit, ACM Transactions on Modeling and Computer Simulation, 16(1):1-25. 2006.

[18] M. Philippsen, A survey of concurrent object-oriented languages. Concurrency: Practice and Experience, 12(10):917-980, 2000.

[19] S.F. Railsback, S.L. Lytinen, and S.K. Jackson. Agent-based simulation platforms: Review and development recommendations. Simulation, 82(9):609–623, 2006.

[20] M. Rettig, Jetlang software Web site. Available from http://code.google.com/p/jetlang/.

[21] S. Srinivasan and A. Mycroft, Kilim: Isolation-Typed Actors for Java. In J. Vitek ed. ECOOP 2008 – Object-Oriented Programming, LNCS, 5142, pp. 104-128, Springer, Berlin ,Germany, 2008.

[22] C. Varela and G.A. Agha, Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Notices, 36(12):20-34, 2001.