Dynamic Distributed Simulation of DEVS Models on the OSGi Service Platform

Martin Petzold, Oliver Ullrich, Ewald Speckenmeyer
{petzold|ullrich|esp}@informatik.uni-koeln.de
Department of Computer Science, University of Cologne
Pohligstraße 1, 50969 Cologne, Germany

Abstract

Interoperability among simulators is one of the key factors in distributed simulations. Several interoperability infrastructures such as HLA and DEVS/SOA have been utilised, but most of them do not provide any dynamics.

This paper introduces the use of the OSGi service platform as universal middleware for dynamic distributed simulation of DEVS models. We have designed and implemented the DEVS/OSGi simulation framework, which is an approach similar to DEVS/SOA, but relies on an integrated service-oriented and protocol independent architecture. It enables standardized plug-and-play capabilities and dynamic reconfiguration within distributed simulations. The architecture and implementation has been validated in an analytical context against a traffic simulation model. We conclude that the standardised interoperability and run-time dynamics provided by the OSGi service platform are highly valuable for distributed simulations.

1 Introduction

Distributed simulations (DS) are used in the context of analytical simulations for the analysis or forecast of the behaviour of a real or imaginary system [6]. In this case detailed quantitative data is collected and simulation runs are typically executed as fast as possible. Further, DS have been applied in distributed virtual environments (DVE), such as Second Life™. In this case the simulation embeds human participants or physical devices and simulation time advances usually in real time. For DVE the need of system accuracy is usually lower than in analytical simulations.

In both domains interoperability infrastructure enables geographically distributed execution and the integration of heterogeneous simulators. Next to proprietary protocols more abstract interoperability infrastructures such as HLA and DEVS/SOA have been utilised for DS of DEVS models [9, 15]. However, in recent work [8, 14] a lack of standardization for plug-and-play capabilities and a need for research on dynamic reconfiguration in distributed simulations has been identified. This should be adopted on an abstract architectural level and possibly also within the simulation nodes.

The DEVS/OSGi simulation framework introduced in this paper addresses both issues. The DEVS standard for modelling and simulation of discrete event systems is being mapped to the OSGi service platform, a widely used industrial standard for modular and dynamic applications in Java. The standardized run-time dynamics and service-oriented (distribution) concepts of OSGi enable plug-and-play capabilities for

distributed simulation. Furthermore dynamic reconfiguration within the distributed simulation environments becomes possible.

In chapter two both standards (DEVS and OSGi) will be introduced, followed by a short overview over recent work in this context. In chapter three we will present the architecture and implementation of the DEVS/OSGi simulation framework. This is followed by some experiments using a traffic simulation model. Finally, some conclusions are drawn and issues for further research are highlighted.

2 Background

2.1 The Discrete Event System Specification (DEVS)

The Discrete Event System Specification (DEVS) is a modular and hierarchical system formalism that was introduced by Bernard P. Zeigler in 1976 [16]. DEVS models consist of atomic and coupled components. Atomic components define a specific behaviour of an entity that is being modelled. Whereas coupled components define the coupling of output to input ports of components. Every atomic component has a defined state at any time. State transition functions prescribe the change of this state if events occur. For internal events the internal transition function and for external events the external transition function define the new state. Furthermore an output function returns the output events and a time advance function returns the remaining time in state.

In 1996 Parallel DEVS was introduced, which allows the parallel execution of DEVS models [2]. Input events are handled as a set of events and a confluent transition function defines the new state in case of concurrent internal and external events. For the simulation of such Parallel DEVS models specific simulation protocols are required. The Parallel DEVS simulation protocol is based on a direct mapping approach and thus the processing of simultaneous internal events. This simulation protocol has been used for our work and is defined by the following steps: 1. Initialization, 2. determination of the next event time, 3. output of events, and 4. execution of state transitions. Even though we focussed on this protocol others could be implemented.

2.2 The OSGi Service Platform

The OSGi standard defines a dynamic module system for Java, the OSGi service platform. The OSGi Alliance (formerly known as the Open Services Gateway initiative) is an industrial consortium (Oracle, IBM, Siemens, ProSyst and others) founded in 1999, which is focused on the maintenance of this standard [10]. Several organizations have implemented the standard. Applications range from industrial automation over mobile applications to enterprise systems. Apache Felix, ProSyst mBS, and Eclipse Equinox (core of the Eclipse IDE) are well known implementations of OSGi.

The OSGi framework is the core of the OSGi service platform. It provides a standard environment for run-time components, called bundles. Bundles are JAR packages with additional manifest headers. Applications are developed from these reusable and loose coupled components. The framework architecture can be divided into three layers (Figure 1). The module layer defines class loading behaviour. In Java it is common to have one classpath, OSGi extends this model with modularization. Thus every bundle has its own classpath and holds private classes, that can be exported to other bundles.

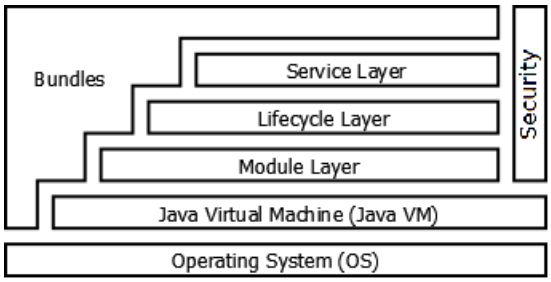

Figure 1: OSGi service platform

The life-cycle layer adds run-time dynamics, that are normally not part of an application. It provides capabilities to install, start, stop, update, and uninstall bundles in an OSGi framework. This allows fine-grained maintenance on the level of bundles.

On the service layer services and the service registry are introduced. The basic model is a 'publish/find/bind/execute' pattern as in a service-oriented architecture (SOA). But it does not rely on web services, in fact in OSGi it operates within an application. Hence, POJOs ('Plain Old Java Objects') can be registered with the OSGi service registry. These services can be found and executed by other bundles.

In the OSGi service compendium [11] and corresponding specifications, such as the OSGi enterprise specification [12], system services for numerous domains of application development are specified. For our work the event admin service and remote services have been used. The event admin service defines a simple 'publish/subscribe' pattern. Thus bundles can register (subscribe) event handler services for a specific topic with the service registry. If an event is posted (published) via an event admin service implementation, it will be delegated to all corresponding event handlers.

Remote services provide the ability to register services not only for internal use in the framework, they can be made remotely accessible, so that bundles of another framework or even other applications can make use of them. Furthermore external services, such as web services, can be registered with the service registry. The type of service distribution and discovery depends upon installed distribution and discovery providers (Figure 2). Several implementations for service discovery exist, e.g. via IP broadcast (Zeroconf or SLP) or distributed configuration servers with Apache ZooKeeper. Distribution providers rely on protocols such as RMI, Web Services (SOAP/REST), or R-OSGi [13].

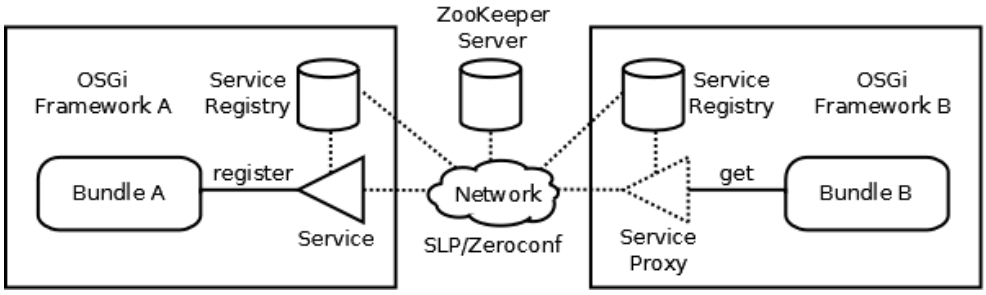

Figure 2: OSGi remote services

2.3 Related Work

Even though OSGi has not been used to simulate DEVS models, some simulation environments use the Eclipse IDE and Eclipse plug-ins for component-based simulation. In [3] the implementation of an object-oriented generic simulation environment based on Eclipse is introduced. The main benefits identified are the separation of model and experiments and the extensibility via Eclipse plug-ins. However, this simulation environment is not capable of simulating DEVS models.

Another flexible, extendible, and reusable simulation framework is James II [7]. It is based on an OSGi-like architecture that uses plug-ins for the integration of different simulators, including DEVS. But this approach does not use OSGi as plug-in environment.

The CD++Builder introduced in [1] is an Eclipse plug-in and thus uses the OSGi bundle concepts, but it is only used in terms of DEVS modelling and does also not use OSGi plug-ins for the implementation and simulation of DEVS models.

Several protocols and distributed middleware concepts have been applied for distributed simulation of DEVS models in Java. The most appropriate one is DEVS/SOA [9], which uses a SOA and thus classic SOAP web services for distributed interoperability among simulators. It enables run-time composability and has been approved of simulating DEVS models in parallel. Even though the architecture of DEVS/SOA is service-oriented, it is still rather different from the service-oriented interoperability and distribution concept provided by the OSGi service layer and OSGi remote services.

The approach of a shared abstract model (SAM) introduced in [15] targets the integration of heterogeneous models and uses proxies as abstraction of model interfaces. SAMs can be implemented using sound distribution middleware and enable flexible and non-tedious integration of new component models. This is indeed close to our intent of the abstraction of the interoperability infrastructure.

Next to distributed interoperability our goal is to enable dynamic reconfiguration within the simulation framework. With a variable structure approach introduced in [8] dynamic reconfiguration can be achieved during run-time. Components can be added, removed, updated or migrated and connections between components can be added or removed dynamically.

However, the run-time dynamics and distribution concepts provided by the OSGi life-cycle and service layer have not been fully utilised yet.

3 The DEVS/OSGi Simulation Framework

The DEVS/OSGi simulation framework introduced in this chapter will enable all benefits of object-oriented application frameworks: Modularity, reusability, extensibility, and inversion of control [5]. Furthermore a service-oriented architecture concept is used for interoperability and extended modularity, e.g. loose coupling and run-time dynamics.

3.1 Approach

Obviously DEVS and OSGi rely on the same basic concept, a decomposed system approach. In DEVS a system is being modelled from decomposed components, while with OSGi applications are build out of components. Furthermore OSGi is based upon service-oriented interoperability concepts, proven to be appropriate for distributed

simulations of DEVS models. Both standards have not been integrated in a well-founded way yet. Thus our approach is to bring together both standards and enable dynamic distributed simulation. In fact two aspects need to be considered, first the mapping of the DEVS formalism onto the OSGi service platform and second the concepts of interoperability and distribution for the Parallel DEVS simulation protocol.

DEVS components (implemented as Java classes) are wrapped into bundles that register a service with the OSGi service registry. Notice, that in case of this 1:1 mapping only one DEVS component is wrapped into a single bundle. Although this is not the best for every application, it enables the full power of OSGi life-cycle layer dynamics, e.g. installation/uninstallation and update of DEVS components. Drawbacks and potential improvements of this mapping will be discussed later.

The interoperability can be described as an abstract integrated service-oriented architecture, that is basically provided by the OSGi service platform. The service-oriented design is similar to the concepts of DEVS/SOA but protocol independent (abstract) and more fine-grained (integrated). The interoperability is abstract as it depends upon the installed discovery and distribution providers. The term integrated reflects the service concept that is being used internally in a simulation node and also externally via OSGi remote services. Both issues will be described in the following two chapters in detail.

3.2 Architecture

The main bundle of the DEVS/OSGi simulation framework (Figure 3) is the DEVS/OSGi core bundle. It consists of all interfaces and abstract classes required for the implementation of DEVS/OSGi components, simulators, and nodes.

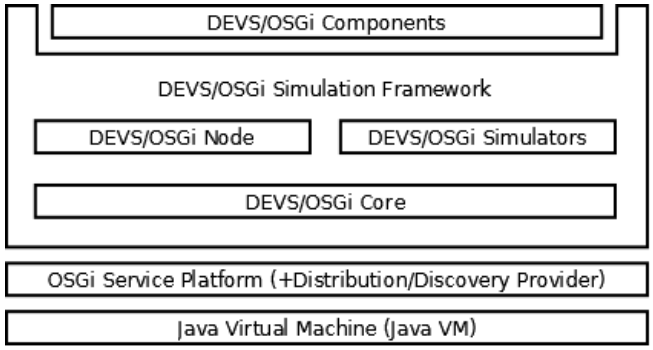

Figure 3: DEVS/OSGi simulation framework

DEVS/OSGi components are simple DEVS component classes that implement a DEVS/OSGi atomic or coupled component interface. These interfaces define the methods of the DEVS formalism and thus should be compatible with every DEVS component implemented for other simulation environments. The component classes are then wrapped into bundles and register a component service on startup (Figure 4a).

DEVS/OSGi simulators implement the steps of the Parallel DEVS simulation protocol. Each simulator installed in a simulation node is responsible for a specific type of component. It instantiates a simulator object for each corresponding component and registers this as remote simulator service with the OSGi service registry (Figure 4b).

DEVS/OSGi nodes are not required for the execution of a simulation, however we have defined a node service that represents our simulation nodes in the service environments. With this it is possible to install, uninstall and update DEVS/OSGi component bundles into remote simulation nodes. Furthermore information about installed components, states and simulation progress are provided that can be used by the controller (GUI).

The interoperability of a simulation is related to simulation control and event delegation. The simulation is controlled by a root simulator that is typically installed in a separate controller node. This root simulator is aware of the simulator service belonging to the root component (this needs to be set/selected). It executes the steps of the simulation protocol via asynchronous calls and dictates the simulation time advance, e.g. real time or as fast as possible. The same is done by every coupled simulator, they are responsible for the execution of the steps on their child simulators. Invocation of all (root/child) simulators is realised via OSGi service layer and OSGi remote services. Thus the simulation control is executed transparently without knowing if the simulator is installed in the local or in a remote simulation node.

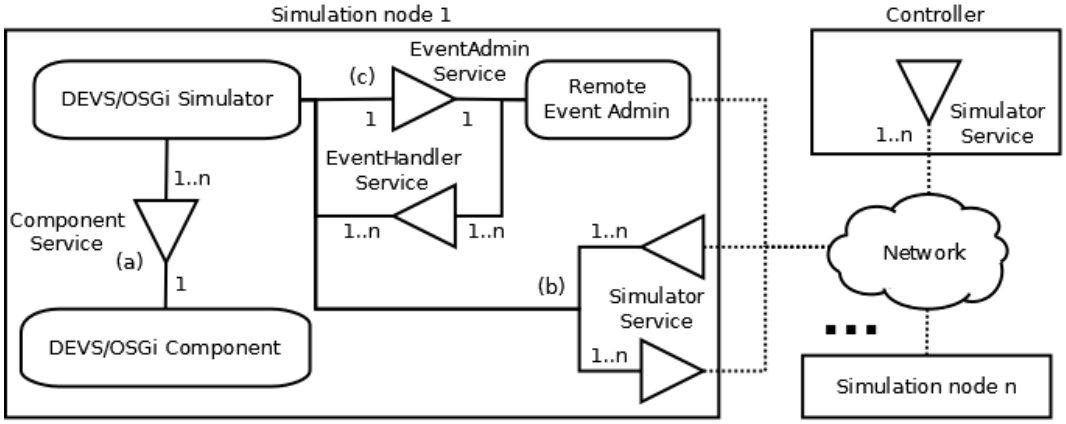

Figure 4: DEVS/OSGi simulation framework services and distribution

Event delegation uses the OSGi event admin and thus a 'publish/subscribe' pattern (Figure 4c). As mentioned before, event handlers are registered for each port. If events are posted to the OSGi event admin, it is responsible for the delegation to all appropriate event handlers of other simulators.

As regular OSGi event admin implementations are only used for framework internal event delegation, we had to implement a remote event admin.

3.3 Implementation

The implementation is similar to others for DEVS, such as DEVS/SOA. However with the implementation of the DEVS/OSGi simulation framework some recent concepts of the Java programming language, such as generics, annotations, and exception handling were integrated. Especially the use of generics for the separation of state and component implementation brings significant improvements for the development of DEVS/OSGi simulators. The state is implemented as separate class and generic type of the

corresponding component. Thus simulators manage the state of its components using Java generics and can store state data via object serialization.

The abstract simulator classes implement a service tracker concept to recognise component services and (child) simulator services installed within the simulation environment dynamically. The event delegation via OSGi event admin has also been implemented on this abstract level. For this we needed to implement a remote event admin implementation, which uses service proxies for remote event handler services and thus uses OSGi remote services for event delegation.

We have implemented two default simulators based on the Parallel DEVS simulation protocol, one is appropriate for atomic, the other for coupled DEVS/OSGi components. The atomic simulator manages the state and functions of components, as mentioned in chapter 2.1. The coupled simulator manages all child simulators and executes the simulation steps. All steps are executed as asynchronous/non-blocking calls.

Furthermore we implemented a default node implementation and a controller (with GUI) for the decentralized installation of DEVS/OSGi components (Figure 5). We built releases of both, node and controller (including the core and simulators), so that they can be used standalone without the Eclipse IDE.

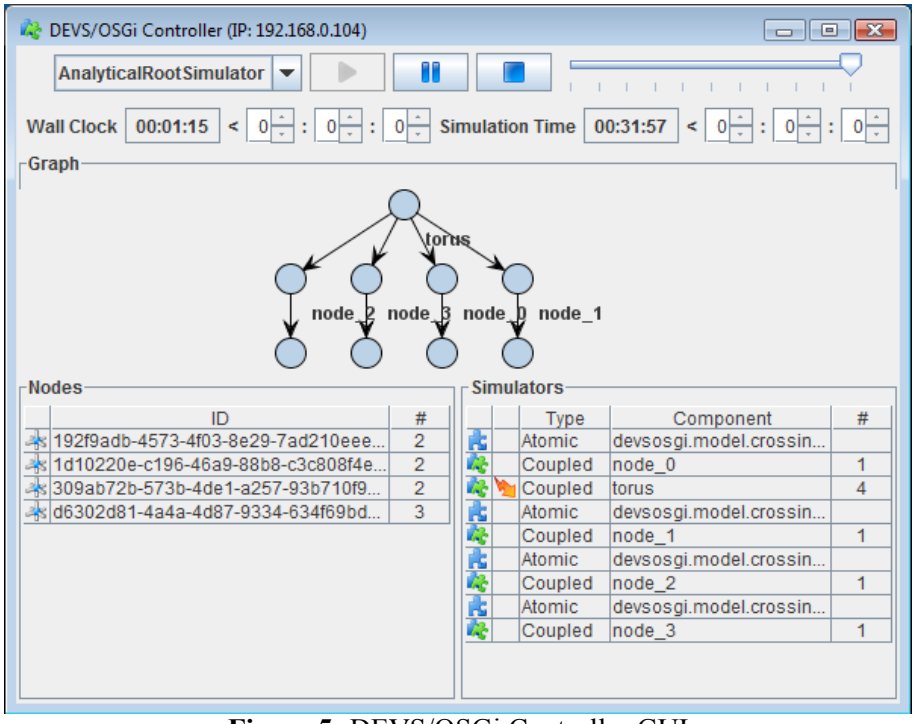

Figure 5: DEVS/OSGi Controller GUI

4 Experiments

In order to validate the framework architecture and implementation we have conducted some experiments. The main objective was to test functionality and performance. For all experiments the DEVS/OSGi simulation framework was installed in

an Eclipse Equinox (3.6.1) OSGi framework. We used the Eclipse Communication Framework (ECF) in version 3.4 as OSGi remote service implementation. The best performance could be achieved with the ECF generic distribution provider based on sockets and ZooDiscovery based on Apache ZooKeeper as discovery provider.

4.1 Traffic Simulation Model

We have implemented a single road crossing as an atomic DEVS component. Cars pass the crossings and are directed to other crossings via output and input events. The traffic lights consist of four states (green, yellow, red+yellow, and red), thus we possibly gain multiple simultaneous events per simulation step. This component was then replicated (wrapped) with different traffic light configurations and cars queuing in front of them. The crossings were connected using coupled DEVS components that form a torus model (Figure 6).

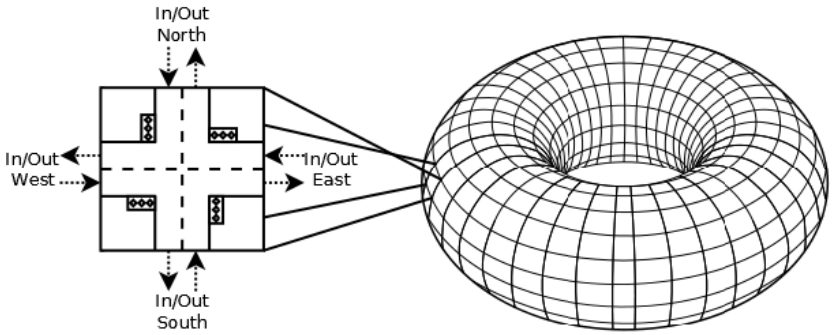

Figure 6: Traffic simulation model

4.2 Results

We have installed models with 4^1 to 4^4 crossings on different distributed simulation configurations (1, 4, 8, and 16 nodes) and run several simulation runs. Every simulation run was set to 24 hours simulation time. The results in Figure 7 show, that we can achieve a speed-up (T1/Tn, Tn = simulation time on n nodes) greater than 1.0 for torus models with 64 and 256 crossings. The torus model with 256 crossings can be executed on four, eight, and sixteen simulation nodes faster than on one. An average speed-up of 1.84 was obtained.

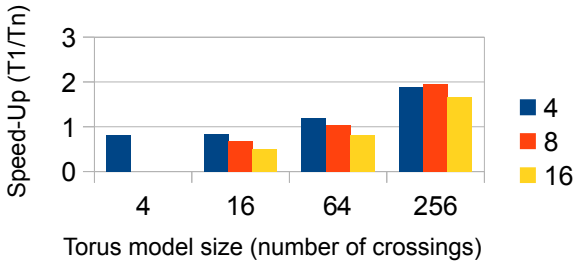

Figure 7: Speed-up on 4, 8, and 16 simulation nodes

4.3 Discussion

The performance tests with the traffic simulation models approved the simulation framework executing steps in parallel. But due to memory utilisation we could install only 256 crossing bundles per simulation node and the installation time was significantly high. For performance tests replication of crossings as bundles is appropriate, but in fact it is not that reasonable installing bundles consisting of the same class into one simulation node. Thus support for multiple occurrence as described in [4] should be considered. In other applications it could be appropriate wrapping more than one DEVS component into a bundle. The challenge is to address these issues without losing the power of the run-time dynamics provided by the OSGi service platform.

5 Conclusions

The OSGi run-time dynamics are highly valuable for plug-and-play capabilities and dynamic reconfiguration in DS, especially for the integration of simulators, components, and distribution/discovery providers, dynamic assembly of distributed models and dynamic distribution of components.

Furthermore OSGi provides a standardised and protocol independent interoperability infrastructure. The DEVS/OSGi simulation framework is fully compatible to Eclipse plug-ins and has been tested using the Eclipse Equinox OSGi framework. Nevertheless some research needs to be done on the implementation of multiple occurrence and the integration of heterogeneous simulators. Further integration into the Eclipse IDE in means of extensions and views would be useful. Additionally OSGi has a lot more capabilities that could be integrated, such as versioning and configuration management.

We focussed our experiments on an analytical simulation context. In DVE the system accuracy is lower and possibly its components more heterogeneous, thus the DEVS/OSGi simulation framework may be even more appropriate.

6 References

[1] *Chiril Chidisiuc, Gabriel A. Wainer:* CD++Builder: An Eclipse-based IDE for DEVS Modeling. Proceedings of the 2007 Spring Simulation Multiconference (2007).

[2] *Alex ChungHen Chow, Bernard P. Zeigler:* Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulator. Transactions of the Society for Computer Simulation International Volume 13, Issue 2 (1996), 55-68.

[3] *Rainer Czogalla, Nicolas Knaak, Bernd Page:* Simulating the Eclipse Way: A Generic Experimentation Environment Based on the Eclipse Platform. Proceedings of the 20th European Conference on Modelling and Simulation (2006).

[4] *Olivier Dalle, Bernard P. Zeigler, Gabriel A. Wainer:* Extending DEVS to Support Multiple Occurrence in Component-based Simulation. Proceedings of the 2008 Winter Simulation Conference (2008).

[5] *Mohamed Fayad, Douglas C. Schmidt:* Object-oriented Application Frameworks. Communications of the ACM Volume 40, Issue 10 (1997), 32-38.

[6] *Richard Fujimoto:* Parallel and Distributed Simulation Systems. New York: John Wiley & Sons, 2000.

[7] *Jan Himmelspach, Adelinde M. Uhrmacher:* Plug'n Simulate. Proceedings of the 40th Annual Simulation Symposium (2007), 26-28.

[8] *Xiaolin Hu, Bernard P. Zeigler, Saurabh Mittal:* Variable Structure in DEVS Component-Based Modeling and Simulation. SIMULATION Volume 81, Issue 2 (2005), 91-102.

[9] *Saurabh Mittal, José L. Risco-Martín, Bernard P. Zeigler:* DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process. SIMULATION Volume 85, Issue 7 (2009), 419-450.

[10] *OSGi Alliance:* OSGi Service Platform – Release 4.2. aQute Publishing, 2009.

[11] *OSGi Alliance:* OSGi Service Platform, Service Compendium – Release 4.2. aQute Publishing, 2009.

[12] *OSGi Alliance:* OSGi Service Platform, Enterprise Specification – Release 4.2. aQute Publishing, 2009.

[13] *Jan S. Rellermeyer, Gustavo Alonso, Timothy Roscoe:* R-OSGi: Distributed Applications through Software Modularization. Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference (2007), 1-20.

[14] *Steffen Strassburger, Thomas Schulze, Richard Fujimoto:* Future Trends in Distributed Simulation and Distributed Virtual Environments: Results of a Peer Study. Proceedings of the 40th Conference on Winter Simulation (2008), 777-785.

[15] *Thomas Wutzler, Hessam S. Sarjoughian:* Interoperability among Parallel DEVS Simulators and Models Implemented in Multiple Programming Languages. SIMULATION Volume 83, Issue 6 (2007), 473-490.

[16] *Bernard P. Zeigler, Herbert Praehofer, Tag Gon Kim:* Theory of Modeling and Simulation. San Diego: Academic Press, 2000.