Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–28

DOI: 10.1177/0037549714533842
sim.sagepub.com
SAGE SCS

An integrated framework for automated simulation of SysML models using DEVS

George-Dimitrios Kapos, Vassilis Dalakas, Mara Nikolaidou and Dimosthenis Anagnostopoulos

Abstract
System models are constructed to design, study, and understand complex systems. According to the systems modeling language (SysML) that is a standard for model-based system engineering, all engineering activities should be performed using a common model. To validate complex system models defined in SysML, simulation is usually employed. There are numerous efforts to simulate SysML models using different simulation methods and tools. However, the efficient support of automated generation of executable simulation code is still an issue tangled by the research community. This paper introduces DEVSys, an integrated framework for utilizing existing SysML models and automatically producing executable discrete event simulation code, according to model driven architecture (MDA) concepts. Although this approach is not simulation-specific, discrete event system specification (DEVS) was employed, due to the similarities between SysML and DEVS, mainly in system structure description, and the mature, yet ongoing research on expressing executable DEVS models in a simulator-neutral manner. DEVSys framework elements include (a) a SysML profile for DEVS, enabling integration of simulation capabilities into SysML models, (b) a meta-model for DEVS, allowing the utilization of MDA concepts and tools, (c) a transformation of SysML models to DEVS models, using a standard model transformation language as query/view/transform (QVT), and (d) the generation of DEVS executable code for a DEVS simulation environment with an extensible markup language (XML) interface. The definition and implementation of DEVSys elements, as well as the process for its application are demonstrated and discussed, with the aid of a simple working example.

Keywords
Simulation methodology, DEVS, SysML, automated code generation, model transformation, simulation tools, MDA, QVT, case study

1. Introduction

In model-based system engineering, as defined by INCOSE,[1] a central system model is used as a reference to perform all engineering activities in the specification, design, integration, validation, and operation of a system. However, most activities are commonly served by autonomous, independently defined system models. For example, system validation is an engineering activity, often being performed in a model-based fashion using simulation.[2] Systems modeling language (SysML) was proposed as a general-purpose graphical modeling language for describing the reference models used to perform all engineering activities for a broad range of systems and systems-of-systems.[3] Specific engineering activities may be accomplished either by the system engineer using a SysML modeling tool (e.g. system design) or by external tools (e.g. system validation) or even by a combination of them. SysML system models should be defined independently of tools, targeting a specific activity, and support different levels of detail to accommodate all engineering activities. However, these models should be properly enriched in order to serve activity-specific purposes, e.g. to facilitate system model validation by an external tool. SysML provides the means to enrich system models, utilizing standard unified modeling language (UML) extension mechanisms, such as stereotypes and profiles.[4]

Department of Informatics and Telematics, Harokopio University of Athens, Greece

Corresponding author:
George-Dimitrios Kapos, Department of Informatics and Telematics, Harokopio University of Athens, 70 El. Venizelou Str, 17671 Athens, Greece.
Email: gdkapos@hua.gr

In the simulation community, simulation methodologies provide the means to define custom system models, which are consequently simulated using corresponding simulation environments. In this case, system modeling and system simulation are not always treated as a unified activity, served by the same system model. Currently, there is a variety of available tools that combine model development and simulation execution. Academic tools, such as CoSMoS or Ptolemy II,[5–7] and commercial tools, such as SimEvents,[8] support component-based modeling and simulation. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. Focus is given on assembling concurrent components. SimEvents, which is an extension of Simulink,[8] provides a discrete-event simulation engine and a component library. Component-based system modeling and simulation (CoSMoS) is a framework aimed at integrated visual model development, model configuration and automatic simulation data collection. The DEVS-Suite is offered as part of CoSMoS,[5] enabling both modeling and simulation according to discrete event system specification (DEVS).[9] Although there are substantial differences in the aim and structure of these tools, such environments facilitate the description of system structure and behavior through a graphical user interface, provide model libraries to ease system description, generate corresponding simulation code, and produce simulation output. Their major disadvantage has to do with the lack of interoperability.[5] In these approaches, system modeling is conducted following the supported simulation modeling method; thus, open standards, such as SysML, cannot be supported for system modeling.

In cases where the system is modelled using SysML, to validate the proposed system architectures, quantitative methods are usually applied, focusing on system performance. Thus, system validation is often performed using simulation. Since most simulation methodologies are model-based, such an approach is suitable for model-based system engineering. This justifies the increased interest in integrating SysML modeling environments and simulation tools. To this end, there are numerous efforts to simulate SysML models.[10–19] Most of them target at transforming SysML system models to simulation models, executed in a specific simulation environment, as for example PetriNets or Modelica.[17,18] Though handled differently, the same issues arise in all these efforts:

(a) The need to embed simulation-specific properties in SysML models, commonly by the definition of a simulation-related profile, is recognized. In order to enable the construction of executable simulation models, simulation-specific capabilities grouped within a SysML profile must be embedded within SysML models to serve system validation.[10,18] These capabilities depend on the simulation methodology applied and should ease the transformation of the SysML model to the corresponding simulation model supported by the selected simulation tool.
(b) The system domain must be clarified. Is the effort targeting a specific domain, as for example production systems,[11] or it may be applied to any system domain? In the first case, the simulation environment usually supports model libraries for the specific domain, thus structural aspects of the model are focused in simulation code generation. In the latter, system behavior should also be defined in SysML and simulation code generation process becomes more complex.
(c) The degree of automation in simulation code generation should be explored. There are efforts resulting in the semi-automation of code generation.[10,16] In this case, the system engineer should be able to actually write code for the specific simulation environment. We argue that code composition should be avoided.
(d) Simulation code generation should be accomplished using standard concepts and methods. Model transformation concepts proposed by model driven architecture (MDA) are explored by current efforts aiming at efficient code generation. A prominent effort in this area is the definition of the SysML4Modelica profile, endorsed by the Object Management Group (OMG),[20] and the corresponding transformations in query/view/transform (QVT) language to convert SysML system models, defined using the profile, to executable Modelica simulation code. QVT is a standard set of languages for model transformation defined by the OMG. The application of the proposed profile is currently under investigation, as corresponding tools were recently made available.[20]

Most of the aforementioned efforts focus on the definition of simulation-related profiles, while the automated generation of simulation code is not fully supported. Furthermore, in most cases simulation model correctness and validity is handled within the simulation environment and not within the SysML model. Here, we argue that simulation-related profiles should provide constraints to ensure system and simulation model correctness before generating the simulation code, minimizing inconsistencies that may occur at this level. Automated simulation code generation should be performed using standard methods and tools, as proposed by OMG.[20] Following MDA guidelines, SysML models may be automatically transformed to executable simulation models. In cases, where domain-specific model libraries are supported by the simulation environment,[19] the transformation is focused on system structure and SysML models are easily annotated with

simulation properties. To promote a more generalized approach, SysML models should be enriched with simulation model behavior description, which in turn, should be transformed into simulation code for a specific simulation framework. In such cases, both the simulation-related profile and the corresponding system model transformation are more complex, to enable model behavior description without requiring existing library components. To this end, the efficient support of automated executable simulation code generation is still an issue tangled by the research community.

This paper presents a successful approach that automatically transforms appropriately enriched existing SysML models to executable DEVS simulation models following MDA concepts,[9,21] as suggested by McGinnis and Ustun,[10] Schonherr and Rose,[11] and OMG.[20] The proposed approach is not restricted to any specific domain, supported by corresponding model libraries. However, it is better suited for systems that can be described in a discrete event context. The existence of a SysML profile, facilitating DEVS model behavior description is a prerequisite and considered as a first step towards this endeavor. As a proof of concept, the preliminary version of the corresponding *DEVS SysML profile* has been presented elsewhere.[22] The presented transformation of SysML system models to DEVS simulation models is considered as the second, equally important step, followed by the execution of DEVS simulation code within existing DEVS simulators. Adopting the proposed approach, simulation models extracted from the predefined SysML models would become executable without any additional programming effort from the system engineer.

Ideally, the engineer would prefer to be relieved from the burden of enriching system models according to the semantics of the simulation framework and the respective profile. However, derived simulation models must be executable. This cannot be ensured by the standard SysML profile, so a simulation-specific profile is required. Additionally, model enrichment is expected to be quite simpler than composing simulation code or recreating a specific simulation model.

DEVS formalism provides a conceptual framework for specifying discrete event simulation models executed on a variety of simulators,[9] such as DEVS-C++ , DEVSJava,[23] cell-DEVS,[24] DEVS/RMI,[25] DEVS XLST,[26] or even DEVS/SOA,[27] which offers DEVS simulators as web services. In any case, executable models are defined either in C++ or Java, while DEVS XLSC accepts as input DEVS models described in XML. To deal with interoperability issues between existing tools, there are considerable efforts to establish a standard XML-based representation for DEVS,[28,29] preparing the ground for the definition of a DEVS meta-model. Moreover, both SysML and DEVS occupy a hierarchical approach in defining model structure, where components may be contained in other components and interconnected with other components through ports.[22] These similarities are exploited in order to integrate them and support the transformation of SysML models to valid executable DEVS simulation models.

Embedding a DEVS formalism detailed description within SysML models enhances their expressiveness in terms of system evaluation, since it enables the straightforward execution of these models on existing simulation environments. At the same time, one might consider that the modeler is restricted when defining system behavior. Thus DEVS-related constraints should be applied only when system evaluation is performed using a DEVS SysML profile that properly extends the SysML metamodel for simulation purposes. SysML models defined using this profile in any standard modeling tool could be consequently simulated in any DEVS simulator.

SysML models can be exported from any standard modeling tool in XML metadata interchange (XMI) format that could be translated to simulation code.[30] Instead of implementing such a translator for each existing DEVS simulation environment, the definition and adoption of a DEVS platform independent model (PIM), described using a meta-object facility (MOF),[31] is suggested here as a fundamental, intermediate transformation stage. MOF is an OMG standard for model-driven engineering that allows the definition of models representing specific domains, such as the DEVS formalism. Building a QVT transformation from a SysML/UML MOF meta-model to a DEVS MOF meta-model enables the transition of models defined in SysML to executable DEVS models in a standard fashion.[32] In this case, SysML models are transformed into DEVS models and consequently are translated to DEVS executable code, using existing tools. A novel MOF metamodel for DEVS makes it usable within standard, open model manipulation tools, such as Medini used for defining and executing QVT transformations. An existing DEVS simulation tool reported by Meseth et al. was chosen for model simulation,[26] as it provided an XML interface for DEVS model description. The XML-based language introduced in the tool is named XLSC and enables the description of DEVS models in XML format, which are consequently executed into a DEVSJava simulator. DEVS SysML Profile and corresponding constraints are implemented in Magic Draw modeling tool.[33] All the required DEVSys framework elements have been implemented, integrated with existing tools and tested.

While SysML-to-DEVS model transformation could be bidirectional (from/to SysML models), in this paper we discuss only the transformation from SysML models to DEVS models. However, the employment of standard tools, languages, and formalisms enables a possible integration of the simulation results back into the SysML system model, following the opposite direction in a similar fashion.

The rest of the paper is structured as follows. A short overview of background work is presented in Section 2, while the DEVSys framework is discussed in Section 3. In Section 4, the DEVS SysML profile is described, emphasizing on the corresponding stereotypes and constraints necessary to enrich SysML models with DEVS model behavior. In Section 5, the DEVS MOF meta-model and the transformation from SysML to DEVS models using QVT is discussed. In Section 6, the necessary transformation and execution of DEVS models in the DEVS XLSC simulation environment is presented. Conclusions and remarks on further research are summarized in Section 7.

A simple textbook DEVS system example is used throughout the paper to facilitate the presentation of the discrete steps of the proposed approach in a comprehensive fashion, facilitating the reader to easily realize model transformations and corresponding tools. Model transformation code listings and screen shots of the employed tools are also included in an Appendix to provide additional information to the interested reader.

2. Background

2.1. Simulating SysML system models

SysML is the system modeling language proposed by OMG, enabling visual modeling of systems and systems-of-systems using a series of standard UML modeling tools. It supports the description of the structure of the system, its behavior and requirements imposed on its operation. Considering the importance of model validation through simulation, the need to integrate SysML modeling tools and simulation environments is evident. To this end, numerous efforts are recorded in the literature from both research and industry communities. In most cases, SysML models defined within a modeling tool are exported in XML format and, consequently, transformed into simulator specific models and forwarded to the simulation environment.

SysML supports a variety of diagrams describing system structure and behavioral aspects, which are commonly required to perform simulation. Depending on the nature and specific characteristics of the system domain under study, there is a diversity of approaches on simulating models defined in SysML, utilizing different SysML diagrams. A method for simulating the behavior of continuous systems using mathematical simulation was reported by Peak et al.,[13] utilizing SysML parametric diagrams which allow the description of complex mathematical equations. System models are simulated using composable objects (COBs).[14] It should be noted that, in any case, SysML models should be defined in a way which facilitates their simulation;[15] thus, simulation-specific characteristics are embedded within system models through corresponding profiles. In the study reported by Paredis et al.,[16] simulation was performed using Modelica. To ensure that a complete and accurate Modelica model is constructed using SysML, a corresponding profile is proposed to enrich SysML models with simulation-specific capabilities, utilizing component constraints and parametric diagram to model system behavior as mathematical equations.[19]

The definition of a SysML4Modelica profile has been endorsed by the OMG.[20] The profile enables the annotation of SysML models with Modelica properties and their simulation in a Modelica simulation environment, utilizing model libraries when available. Operational QVT language is adopted to transform SysML models to Modelica models. Corresponding specifications and tools have recently become available for usage by the community.

The aforementioned approaches are suited for system domains simulated using models with continuous behavior. However, simulation of discrete event systems is also feasible based on SysML system models, where system behavior is described in activity, sequence or state diagrams. In the study by McGinnis and Ustun,[10] system models defined in SysML are translated to be simulated using Arena software. SysML models are not enriched with simulation-specific properties, while emphasis is given to system structure rather than system behavior. MDA concepts are applied to export SysML models from a UML modeling tool and, consequently, transform into Arena models, which should be enriched with behavioral characteristics before becoming executable. This may be accomplished by the system engineer either with the use of existing model libraries within the Arena tool, or with Arena-specific simulation code composition. In the study by Batarseh and McGinnis,[34] the SysML4Arena profile was introduced for the specification of system models that are exported and transformed via ATL to Arena simulation models. This approach emphasizes domain-specific languages and existing library components.

In the study by Wang and Dagli,[17] the utilization of Colored Petri Nets is proposed to simulate SysML models. If the system behavior is described using activity and sequence diagrams in SysML, it may be consequently simulated using discrete event simulation via Petri Nets.

However, none of the aforementioned approaches, targeting discrete event simulation, supports the fully automated generation of simulation code, executed in the corresponding simulation environment, especially when model libraries are not available. Most of them adopt the definition of a simulation-related profile to enrich SysML models with properties necessary to simulate them according to the adopted simulation methodology,[10] especially in cases where model behavior is described in SysML.[17] Moreover, adoption of model transformation tools based on MDA to produce simulation models from SysML system models, gains momentum. The main hinder for implementing such an approach is the lack of standard meta-models for simulation methodologies.

2.2. DEVS framework and tools

DEVS models can be simulated in a series of appropriate simulation environments. However, executable DEVS models cannot be derived from pre-existing system models and in most cases they have to be defined in terms of code, independently of the system model.

According to DEVS formalism,[23] mathematical sets are introduced to describe the structure and behavior of a model. Models are specified in a modular and hierarchical form, while two types of them are defined: *atomic models*, focusing on behavioural aspects, and *coupled models*, focusing on structural aspects, thus expressing how atomic and other coupled models are connected in a hierarchical form to build more complex structures.[9]

Each atomic model is described as:

- a set of input ports for receiving external events
- a set of output ports for sending external events
- a set of state variables and parameters, used for model states' definition
- an internal transition function (*deltint*), which specifies the next state to which the system will transit
- an external transition function (*deltext*), which specifies the next system state when an input is received, computed on the basis of the present state, the elapsed time, and the content of the external input event
- an output function (*lambda*), which generates an external output just before an internal transition occurs if required
- a time advance function (*ta*), which controls the timing of internal transitions.

A coupled DEVS model contains:

- a set of components
- a set of input and output ports
- an external coupling, connecting the input/output ports of the coupled model to one or more input/output ports of its components
- an internal coupling, connecting output ports of the components to input ports of other components

There is a variety of DEVS simulators developed by different groups. In practice they all provide a set of libraries to be used in C++ or Java languages. As they are independently developed, the need for interoperability between them arises. Thus, many DEVS simulation environments aim at the definition of XML-based DEVS model description and their corresponding interpretation in different programming languages.[26,28,29,35,36] An XML data encapsulation was accomplished by Hosking and Sahin,[35] within the DEVS environment, as a unifying communication method among the entities in any Systems-of-Systems (SoS) architecture. In the study by Mittal et al.,[36] the problem of model interoperability was addressed, introducing DEVSML. The composed coupled models are then validated using atomic and coupled document type definitions (DTDs). In this case, model behavior is not emphasized.

In any case, executable models are either in C++ or Java. The proposed XML representations focus on low-level description of executable DEVS code, including language commands and variable value assignments. As reported by Meseth et al.,[26] XLSC DEVS was introduced as an XML language for modeling atomic and coupled DEVS models. It was shown that (a) XLSC can express a model's behavior as well as its structure, and (b) an XLSC model can be simulated. A prototype interpreter was implemented in Java and employed to directly execute the model. In this case, atomic model behavior can be described in XML using a series of low-level, yet programming language independent actions, denoting specific instructions.

In the study by Risco-Martín et al.,[28] DEVS-XML was proposed as a platform-independent, XML-based format for describing DEVS models. DEVS-XML is consequently transformed into executable code for existing DEVS simulators, using translators, such as the ones proposed for a DEVSJava simulator,[28] which was only implemented for DEVS coupled models. Thus, DEVS behavior transformation is missing. DEVS-XML was proposed to establish DEVS model mobility and promote interoperability between discrete DEVS simulators, independently of the programming language they are implemented in and the way they operate (either in a distributed or centralized fashion). However, there are not many tools supporting it yet. Among them, the one reported by Risco-Martín et al. is the most advanced,[29] and is currently used to transform DEVSJava code into XML and vice versa, for a subset of DEVS formalisms, FD-DEVS.[37] The latter DEVS-XML version is referred as XFD-DEVS and offers XSD definitions along with a tool for the transformation.

DEVS-XML offers an upper level representation of DEVS behavior based on system state transitions, compatible to SysML state machine diagrams. Thus, its adaptation as an intermediate DEVS representation format was explored. A few issues were raised. First, DEVS-XML does not incorporate the relationship between state variable values and states. Second, in order to be implementation independent, DEVS-XML is quite general and, therefore, does not handle complex expression values in a specific manner. Third, DEVS-XML is -as implied by its name- the specification of a flat XML representation for DEVS models, rather than a meta-model for -conceptually richer- DEVS models. Fourth, in DEVS-XML, XML attributes are rarely used, leading to element explosion and quite complex structure. Finally, actual testing and use of DEVS-XML compliant tools is rather difficult.

Therefore, there is a need for a standard representation for DEVS models that is (a) consistent with DEVS theory, (b) ready to be executed in DEVS simulation environments, and (c) compatible with SysML/UML meta-model, (i.e. defined according to OMG's MOF) to facilitate the transformation of SysML to DEVS models using standards-compliant, existing tools. Such a meta-model may take advantage of all the work already performed on XML representations of DEVS models.

3. DEVSys Framework

Both SysML and DEVS are established and widely referenced in the areas of system modeling and simulation, respectively. In this context, the *DEVSys framework* is defined for simulating SysML models using DEVS simulators, provided that models are described in a way compatible to the DEVS formalism. The system engineer specifies his/her system model using SysML, via a UML modeling tool and receives valid simulation results from the execution of the corresponding DEVS model in a specific simulation environment. Thus, fully automated DEVS simulation code generation is provided. An overall perspective of the framework is depicted in Figure 1.

In order to enable simulation, the system engineer should enrich SysML system models with DEVS-compliant simulation information. This is enabled by the proposed DEVS SysML profile, consisting of a set of stereotypes and constraints. Stereotypes are employed to characterize specific SysML model elements as DEVS-related, used to create DEVS executable models. Constraints restrict SysML models according to DEVS formalism and validate their correctness and completeness within the system modeling tool, so that they can be automatically transformed to executable DEVS simulation code. These activities constitute the first step of the proposed process and are conducted using a UML modeling tool, such as Magic Draw or Visual Paradigm, using SysML and DEVS SysML profiles.

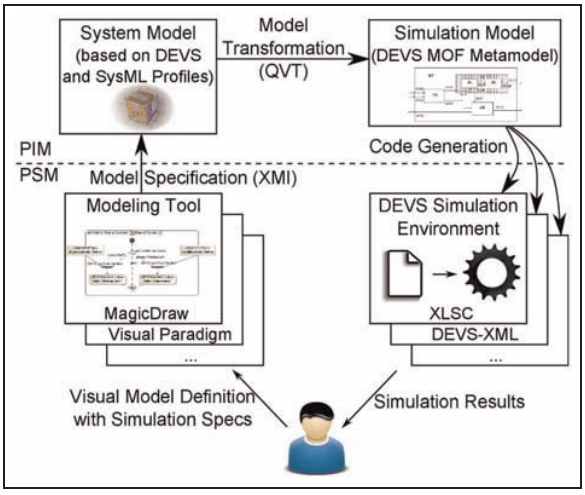

Figure 1. DEVSys: Automated SysML model simulation using DEVS.

The SysML model defined in any standard UML modeling tool may be exported in XMI format. The XMI representation of models is used to ensure interoperability between different modeling tools and is implementation independent. This is the second step of the proposed approach.

Since DEVS formalism is also supported by numerous implementations, the SysML to DEVS model transformation includes an intermediate, yet autonomous and very important step, which is the generation of a pure, implementation independent DEVS representation of the system model, based on a DEVS MOF 2.0 meta-model. The existence of a DEVS meta-model independent of specific simulators enhances the usability of the proposed approach and facilitates simpler transformations for diverse simulation environments, such as DEVSJava, cellDEVS, etc. Therefore, such a DEVS MOF meta-model is introduced.

Transformation of DEVS enriched SysML models to DEVS model representations is specified and implemented using QVT. Specifically, QVT defines three transformation languages: QVT-Operational, QVT-Relations and QVT-Core. In this case, QVT-Relations, a declarative language for defining constraints on source and target model elements, has been used. QVT transformations can be applied on models that conform to MOF 2.0 meta-models (SysML and the introduced DEVS meta-model in our case). Object constraint language (OCL),[38] another OMG standard language for defining constraints, is integrated and also extended in QVT with imperative features. This constitutes the third step of the proposed approach.

The fourth step constitutes of the transformation of DEVS models to executable code for specific DEVS simulation environments and is feasible, since all DEVS related information is contained in the DEVS models. However, this transformation depends on the target simulation environment and is implementation specific. In our current implementation, XLSC DEVS was used as the simulation environment.[26] The simulator accepts as input DEVS models described in XML and simulates them in a Java environment.

From a software engineering perspective, according to the concepts of MDA, for each real-world domain, two kinds of discrete models should be defined: a PIM (platform independent model), ensuring proper domain representation, and PSMs (platform specific models), corresponding to one or more executable versions of the PIM. In the DEVS simulation domain, DEVS MOF meta-model is used to define a PIM, and is consequently translated into code executable on a variety of DEVS simulators, such as XLSC DEVSJava and DEVS/SOA,

corresponding to PSMs. In the SysML modeling domain, DEVS SysML profile is used to define SysML models enriched with simulation capabilities. The SysML meta-model and DEVS profile are used to define a PIM, while discrete UML modeling tools internal representations correspond to PSMs (see Figure 1).

From an implementation perspective, the DEVS SysML profile and a corresponding application programming interface (API) is implemented for MagicDraw,[33] which is a widely used UML modeling tool supporting SysML with a user-friendly programming interface. SysML models being enriched, using the DEVS SysML profile and exported in XMI format are transformed into DEVS models (conforming to the DEVS MOF meta-model) via a QVT model transformation, that has been implemented for that purpose, using Medini tool (http://projects.ikv.de/qvt/). As a proof of concept, the last part of the transformation (code generation) has been implemented for XLSC DEVS execution environment in terms of eXtensible Stylesheet Language Transformations (XSLT).[39] Although there is a single DEVS SysML to DEVS transformation (defined with QVT), usable in all cases, a distinct final transformation would be required for each DEVS simulator that is intended to be integrated in the framework. In any case, the transformation from DEVS SysML to DEVS and the transformations from DEVS to simulator-specific formats are defined once and can be used for all system models in the following.

Within DEVSys framework, the main contribution is related to (a) the definition of the DEVS SysML profile and the corresponding implementation for MagicDraw modeling tool, (b) the definition of the DEVS MOF 2.0 meta-model, and (c) the definition and implementation of the QVT transformation of DEVS SysML PIMs to DEVS PIMs. An XSLT transformation of DEVS PIMs to XLSC DEVS PSMs has also been implemented as a proof of concept.

Additionally, the model-based nature of the approach has a positive impact on the quality of the generated executable simulation model. Since it is automatically generated from the actual system model, syntactic errors are eliminated, while semantic ones are more easily avoided, due to the higher level provided for the specification of model structure and behavior (compared to code composition). Specification and implementation of the above-mentioned framework elements are described in the following sections.

To explore the steps of the proposed framework, a simple textbook example, widely used in DEVS literature, is employed and discussed in the following sections. It consists of a simple processor model and its experimental frame (EF), indicating the conditions under which the processor operates. The overall system is called EFP and is described in Figure 2. The system consists of a processor, which is an atomic DEVS model and the EF, coupled

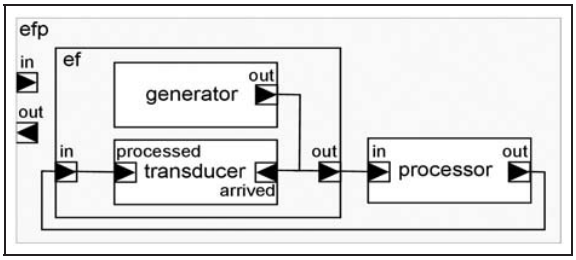

Figure 2. EFP DEVS model.

DEVS model, consisting of a generator of requests directed to the processor and a transducer collecting statistics. The simplicity of the system enables the detailed description of all the steps required from defining the system model using SysML and converting the SysML model to executable simulation code within the context of the paper.

4. DEVS SysML Profile

As indicated in Figure 1, system modelers initiate the process of automated SysML model simulation by specifying system models, enriched with simulation properties. DEVS SysML Profile is the framework element that defines the appropriate extensions and constraints for SysML models, so that DEVS simulation capabilities are successfully integrated into system models. Additionally, profile constraints ensure system model validity and, therefore, minimize inconsistencies in the derived simulation model and executable simulation code.

In the study by Nikolaidou et al.,[22] the notion of a DEVS SysML profile has been introduced. In this section, we present a detailed version of the profile, as it has been extended and evolved to serve the implementation of the proposed framework. The current version of DEVS SysML profile is more fine-grained, depicting detailed aspects of structural (ports, composition) and behavioral (DEVS functions) elements and their relationships.

DEVS SysML profile has been implemented in MagicDraw modeling tool. Proposed stereotypes are defined using standard tool interface, while constraints are implemented using OCL, model customization and the API provided. To better understand DEVS-related stereotypes defined in the profile, similarities between SysML and DEVS are outlined, before discussing profile definition and usage based on EFP example (Figure 2).

4.1. Similarities between SysML and DEVS

The main structural elements of system description in SysML are *blocks*, which may contain value properties (variables), part properties (other contained blocks), reference properties (references to other blocks), ports, used as the endpoints of inter-block connections, and constraints,

indicating relations between block properties. Ports facilitate sending or receiving events (standard ports) or data items (flow ports).

System structure is defined via *block definition diagrams (BDDs), internal block diagrams (IBDs)*, and *parametric diagrams (PDs)*. BDDs provide an overall hierarchical representation of system structure and composition. IBDs focus on composite block internal description, where component block relations are specified. Although BDDs define the sub-components of a block, an IBD defines how these components are interconnected through ports. PDs bind specific block properties to block constraint variables, enabling -this way- constraint verification and/or enforcement.

System behavior is described using *State Machine Diagrams (SMD)* (defining block states and state transitions), *activity diagrams (AD)* (emphasizing on the actions performed by blocks), *sequence diagrams* (SD) (emphasizing on synchronization of block actions and produced or received events) and *use case diagrams* (UC) (describing use cases of the blocks). Finally, the *requirement diagram (RD)* is introduced to define system requirements. As reported by Nikolaidou et al.,[22] we have identified a set of correspondences between DEVS and SysML modeling elements. Coupled and atomic DEVS models correspond to SysML blocks, while DEVS ports correspond to SysML flow ports. DEVS state variables correspond to SysML block value properties, while SysML constraints may be used to depict the way state variables are interrelated to indicate system states. DEVS coupled model description, i.e. component models interconnection, is similar to the SysML IBD diagram, corresponding to each composite block. DEVS coupled component models can be expressed as SysML part blocks. In both cases, either part blocks or coupled model component communication is depicted using connections between input and output ports.

However, simulation execution depends on model behavior, apart from system information and structure. Therefore, atomic DEVS functions should be expressed in terms of SysML. Internal transition function defines state transitions, making SMD as the evident selection for this purpose. Additional characteristics of SMDs, such as duration guard conditions and effects on state transitions, make this diagram type appropriate for defining output and time advance functions, as well. Nevertheless, these functions are strongly related to states and state transitions. On the other hand, external transition functions are triggered by external events and result in state and/or variable modification. This could be clearly described in activity diagrams.

Table 1 illustrates the correspondence between DEVS and SysML entities. The structure of a system in both SysML and DEVS is defined in a similar fashion, allowing the mapping between SysML and DEVS models.*22* DEVS profile should emphasize on DEVS atomic model behavior, utilizing State Machine, Activity and Parametric diagrams.

Table 1. Mapping between DEVS formalism and SysML entities.

DEVS formalism	SysML entity
Atomic and coupled model	Block
Input port	Flow port with 'in' direction
Output port	Flow port with 'out' direction
Atomic model	Block
State variables	Value properties and constraints
Parameters	Value properties
DEVS atomic model functions (*deltint, deltext, lambda, ta*)	Behavior diagrams (State machine, activity)
Coupled model	Internal block diagram
Component models	Block parts
Internal coupling	Connectors between flow ports of IBD's parts
External coupling	Connectors between flow ports of the IBD's enclosing block and its parts

4.2. DEVS SysML profile diagrams

The formal method proposed by the OMG for extending or restricting SysML/UML, so that a specific domain, such as DEVS formalism, may be effectively modeled, is the definition of stereotypes grouped by means of a profile.[4] Using DEVS specific stereotypes for SysML behavior diagrams, their functionality can be restricted to conform to DEVS formalism (e.g. the description of DEVS atomic model functions). Such a profile must facilitate the following:

- Ensure that the structure of system models defined in BDDs and IBDs contain all necessary information to simulate them using DEVS. This may be accomplished by specific constraints checking the ports and properties, defined for system blocks participating in BDDs and IBDs to ensure that all DEVS related information is provided.
- Provide the means to characterize specific blocks as DEVS atomic models and provide DEVS SysML diagrams to describe DEVS functions used to define the behavior of atomic simulation models.

4.2.1. DEVS coupled model. DEVS models can be specified only in SysML system models described using BDDs. System blocks (with unidirectional ports) can be identified as DEVS blocks and categorized as either *DEVS Coupled* or *DEVS Atomic* blocks. A *DEVS Coupled* block consists of a set of other blocks (atomic or coupled) and a coupling element, expressed as an internal block diagram (IBD). The SysML block diagram corresponding to the EFP

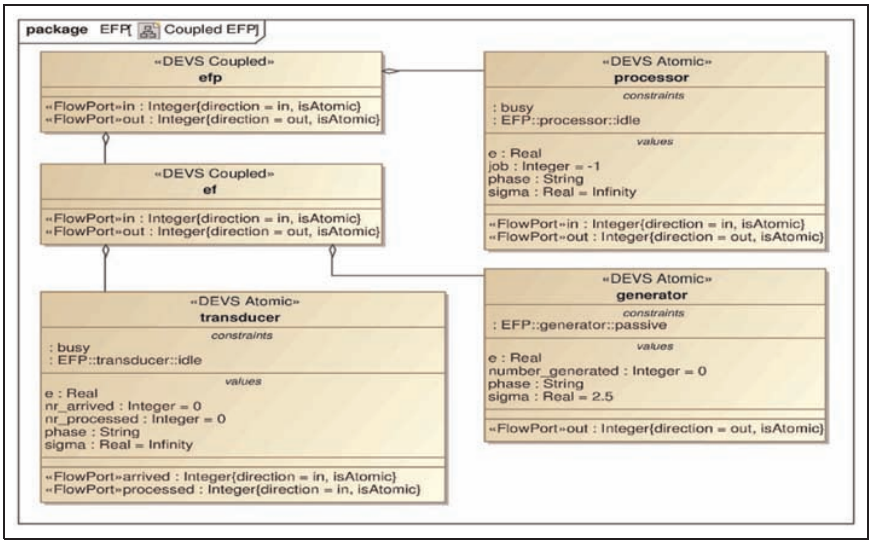

Figure 3. EFP SysML block definition diagram.

Table 2. DEVS model structural stereotypes.

DEVS stereotype	SysML entity	Constraints
DEVS model	Block definition diagram	There is one BDD containing the overall model. Only DEVS Coupled and DEVS Atomic entities participate in this diagram.
DEVS Coupled internal diagram	Internal block diagram	A DEVS Coupled Internal Diagram must be associated to any DEVS Coupled. The diagram may only contain the DEVS models directly used by the corresponding DEVS Coupled. All flow ports must be connected in an appropriate manner (output to input flow port).
DEVS Coupled	Block	A DEVS Coupled may only have property parts and unidirectional (in/out) flow ports. Every DEVS Coupled is associated to a DEVS Coupled Internal Diagram.
DEVS Atomic	Block	A DEVS Atomic may only have unidirectional (in or out) flow ports, value properties and constraints on the value properties. Four sub-diagrams must be associated to each DEVS Atomic to describe its behavior: DEVS States Definition, DEVS States Association, DEVS Atomic Internal and DEVS Atomic External.

model of Figure 2 is depicted in Figure 3. It consists of the *processor* and the *ef* block. The latter consists of a *generator* and a *transducer* block. Constraints, properties and ports are shown for each block. All blocks are annotated by either *DEVS Coupled* or *DEVS Atomic* stereotypes. So far, no specific DEVS-related entities are defined.

The coupling of a *DEVS Coupled* block defines the interconnections between (a) the ports of part blocks (internal connections) and (b) the ports of the container *DEVS Coupled* block and its parts (external connections). All this information is included in the corresponding SysML IBD. When the SysML block is characterized as a *DEVS Coupled* block, related DEVS structural constraints, as defined in Table 2, are applied to ensure that all couplings between container and part block port are properly defined. The IBD in Figure 4 shows port interconnection for contained blocks of the *ef* complex system (coupled model). No specific DEVS stereotypes are used to describe system model. Thus, *DEVS Coupled* block constraints are applied to ensure that all necessary port connections are defined.

4.2.2. DEVS atomic model. DEVS-related entities should be defined for any SysML block characterized as *DEVS Atomic* block, by applying the corresponding stereotype, in order to describe simulation model behavior. To describe atomic model behavior, system states and the four related functions, namely *deltint, deltext, lambda*, and *ta*, should be defined. When the DEVS Atomic stereotype is applied on a system block in a BDD diagram, DEVS structural constraints are also applied to the specific block to ensure the definition of DEVS ports and state variables.

Table 2 contains DEVS SysML stereotypes, SysML/UML entities on which they are applied, and constraints, regarding model structure for the definition of DEVS Coupled and Atomic blocks.

DEVS Atomic model behavior is defined as transitions between discrete model states.[9] Thus, a set of states and simulation model behavior must be described for any DEVS Atomic block, using a series of diagrams and the corresponding DEVS stereotypes. For this purpose, four sub-diagrams must be related to each Atomic DEVS Block. Two of them facilitate state definition and the other two function definition:

- *DEVS State Definition Model*: A SysML constraint BDD defining constraints, each of them denoting a possible system state.
- *DEVS State Association Model*: A PD that facilitates state definition based on the constraints of the previous DEVS State Definition Model. The states (constraints) are formed from their association with state variables (value properties).
- *DEVS Atomic Internal Model*: A SMD facilitating the definition of internal transition function, output function and time advance function.
- *DEVS Atomic External Model*: An AD facilitating the definition of external transition function.

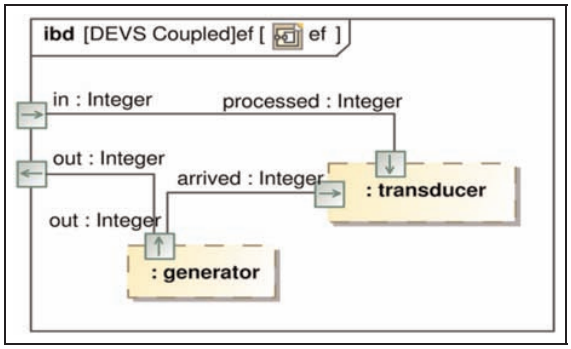

Figure 4. EF SysML internal definition block.

The *processor* Atomic Block (see Figure 3) is presented in the following to discuss how Atomic Blocks are modeled. It receives a job number (*FlowPort in*), processes it and finally outputs the job number (*FlowPort out*). The block is described by four properties, namely *e, job, phase*, and *sigma*, which may be used as DEVS state variables, as discussed in the following. Property type and initial values are also defined in a standard SysML fashion. As shown in the Figure 3, constraints are also defined for *processor* Atomic Block. They relate to DEVS state definition and are also discussed in the following.

4.2.3. DEVS State Definition and Association. DEVS atomic model behavior is described as transitions between valid states, thus, state definition for each DEVS Atomic block should be facilitated. Each state can be considered as a combination of state variable values, while state variables are defined as block value properties. Thus, the state variable value properties are associated to the DEVS State constraints in the corresponding SysML PD. DEVS states are defined in the *DEVS State Definition Model* as *DEVS State* constraints and are further explained as combinations of state variable values in the *DEVS State Association Model*. The latter, a stereotype of the PD, is used to show the constraint each *DEVS State* enforces on *State Variable* values. Table 3 contains related DEVS SysML stereotypes, corresponding SysML entities and constraints.

As an example, states definition of *processor* block is presented in Figure 5. The *processor* is either *idle* or *busy*. The corresponding state variable, named *phase*, is introduced and associated with the state constraints in the DEVS State Association Model of Figure 6. Defined as a value property of *processor* block, it is characterized as state variable, once added in DEVS State Association Model. It may be assigned to one of either two values: *idle, busy*.

As shown in Figure 5, state variables are represented as constraint blocks in DEVS State Definition Model. Each state variable is associated with one or more parameters (*s*) used to define it.

Table 3. DEVS state definition stereotypes.

DEVS stereotype	SysML entity	Constraints
DEVS states definition	Block definition diagram	It must be associated to a DEVS Atomic Block. The diagram may only contain DEVS State Constraints.
DEVS states association	Parametric diagram	It must be associated to a DEVS Atomic Block. The diagram contains the block's value properties, the DEVS State Constraints (defined in DEVS States Definition Diagram) and their interconnection. Each constraint parameter must be connected to a value property.
DEVS State	Constraint	It may have as many parameters as the number of state variables. The type of each parameter must be compatible to a subset of the state variable's type. The value of each parameter must be constrained.

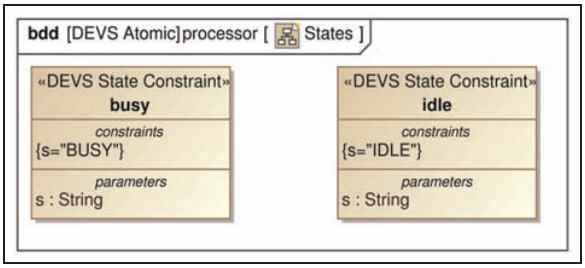

Figure 5. Processor DEVS state definition model.

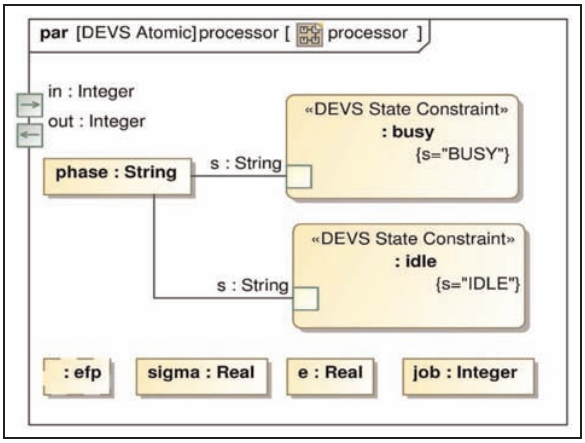

Figure 6. Processor DEVS state association model.

The corresponding DEVS State Association Model, indicates the exact way these parameters are associated with corresponding value properties characterized as state variables. As shown in Figure 6, the *s* parameter of DEVS State constraint *busy* is associated to the *phase* value property of the *processor* atomic DEVS block. Consequently, discrete *processor* states could automatically be inserted in DEVS Atomic Internal and DEVS Atomic External diagrams defining DEVS atomic model functions.

4.2.4. DEVS Atomic Internal Model. The *DEVS Atomic Internal Model* of a *DEVS Atomic* block specifies the behavior of the atomic DEVS model in case of internal state transition. Therefore, it is declared as a stereotype of state machine diagram (SMD), where *DEVS Internal Transitions* between the *DEVS States*, already defined in *DEVS State Definition Model*, occur at predefined *Time Advances* and may produce *DEVS Output* to some of the output ports of the atomic block. In DEVS, an internal transition function specifies the next state to which the system will transit. DEVS states are computed based on DEVS State Definition Model and are automatically inserted in DEVS Atomic Internal Model. The system modeler specifies DEVS Internal Transitions by inserting

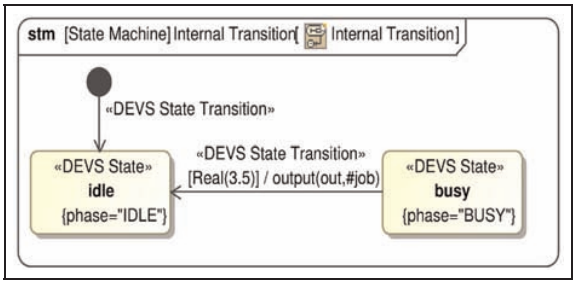

Figure 7. Processor DEVS atomic internal model.

DEVS State Transitions between DEVS States. Output function generates an external output just before an internal transition occurs and Time Advance function controls the timing of internal transitions. The initial state is determined by the initial values of each state variable.

The DEVS Atomic Internal Model for *processor* block is depicted in Figure 7. Atomic model *processor* changes its state from *busy* state to *idle* (as defined by DEVS Internal Transition represented as a state transition) after 3.5 seconds (as defined by Time Advance, represented as Timing Condition of the corresponding transition). It also produces output, as indicated by transition effect representing DEVS Output. Specifically, job number is assigned to *out* output port. Table 4 contains related stereotypes and constraints. As there is no outgoing transition from *idle* state, this state can change only in the occurrence of an external event.

4.2.5. DEVS Atomic External Model. The *DEVS Atomic External Model* of a *DEVS Atomic* block specifies its behavior in case of external state transition caused by the arrival of a specific input. It is declared as a stereotype of activity diagram (AD), where possible *DEVS States* and *DEVS Input* are combined. Each combination results in a set of actions, where (simple or state) variables of the atomic block may be modified. This function is executed whenever an input event arrives at the atomic DEVS model, e.g., an input port receives a specific value. Therefore, there is a condition (of stereotype DEVS Input) for every distinct value received by an input flow port of a DEVS Atomic block. The corresponding actions (of stereotype DEVS Variable Modification) are also determined by the state of the atomic DEVS model at the time of the arrival of the input event. Thus, initially there is a DEVS State Var Check decision node, checking current state variable values and creating different control flows. Corresponding stereotypes and constraints reside in Table 5. The DEVS Atomic External Model for *processor* block is depicted in Figure 8, as the working diagram of the corresponding MagicDraw screen shot. As shown in the figure the stereotypes corresponding to this diagram

Table 4. DEVS atomic internal stereotypes.

DEVS stereotype	SysML entity	Constraints
DEVS atomic internal model	State machine diagram	The diagram must be associated to a DEVS Atomic Block. The diagram contains an initial node, the DEVS States as derived from DEVS State Definition Diagram, transitions from state to state and notes indicating Time Advance.
DEVS state	State	Each state must be defined as a state constraint in the DEVS State Definition Diagram of the same DEVS AM.
DEVS internal transition	State transition	Only one transition may start from any single state node.
DEVS OutFn	DEVS internal transition	The action body of the transition has value assignments to the output flow ports of the DEVS Atomic Block.
DEVS *Ta*	Note	The DEVS *Ta* note is associated to a DEVS Internal Transition state transition. The note contains the mathematical function describing advancement of time describing *Ta*.

Table 5. DEVS atomic external stereotypes.

DEVS stereotype	SysML entity	Constraints
DEVS atomic external model	Activity diagram	It must be associated to a DEVS Atomic Block. The diagram contains an initial node, a DEVS State Check, DEVS In Ports (equal to the block's input ports), DEVS State Forks, DEVS State Input Joins, DEVS State Modification Actions and final nodes.
DEVS state check	Decision node	The guard conditions starting from this node check state variables and end at DEVS State Forks or DEVS State Input Joins.
DEVS in ports	Parameter	May only connect to a DEVS State Input Join.
DEVS state forks	Fork/Join	Is connected to one or more DEVS State Input Joins.
DEVS state input joins	Fork/Join	Is connected to a DEVS State Modification Action or a final node.
DEVS state modification actions	Action	The Action assigns a value at a value property of the block. Is connected to another DEVS State Modification Action or a final node.

may be used from the palette. Input must be received from the *in* port and at the same time the state must be *idle*. This results in modifying the state to *busy*. If one of these conditions is not satisfied, nothing happens. This example shows that different input and state combinations result in distinct new model states. Execution of external transition for each received event does not consume any simulation time.

4.3. DEVS SysML profile diagram

The corresponding class diagram of the proposed DEVS profile for SysML is depicted in Figure 9. DEVS formalism entities are depicted as stereotypes of SysML entities. Associations between entities included in the class diagrams are aggregations (as in the case of *DEVS states* defining a *DEVS Internal Transition*), compositions (as in the case of *DEVS Model* composed of *DEVS blocks*), generalizations (as in the case of *DEVS Coupled* block defined as a descendant of *DEVS* block) and generic associations, where textual description is given. Multiplicity constraints are also used. All DEVS SysML stereotypes defined, their properties and the relations between them are included in the system models exported in XMI format from MagicDraw modeling tool, since extensions are defined in a standardized fashion.

5. SysML to DEVS Model Transformation

Having defined the DEVS SysML profile, construction of system models with simulation characteristics that can be exported in XMI format is feasible. DEVS SysML models comply with the UML2 meta-model, which is a quite general meta-model that can be used for modeling a variety of artifacts, systems, processes, etc. This means that, although we focus on the DEVS-related information within the

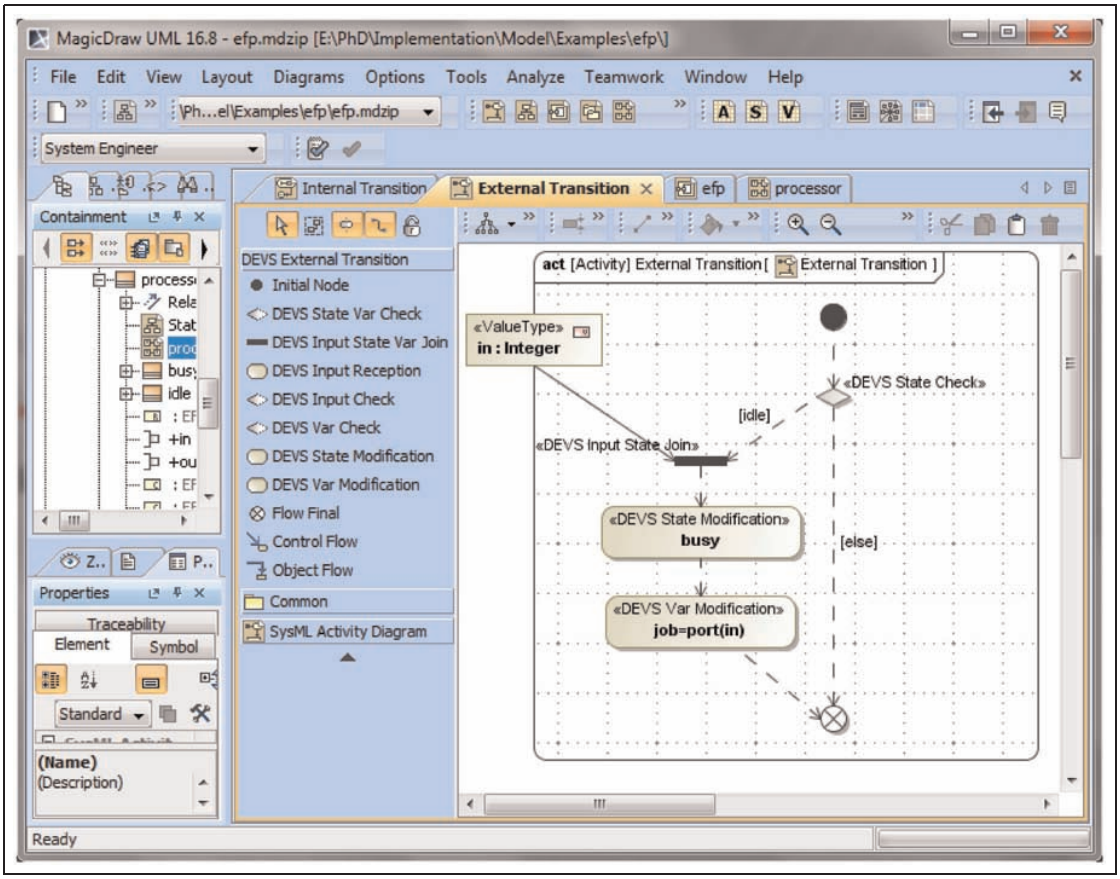

Figure 8. Processor DEVS atomic external model.

DEVS SysML profile, the entities of the models contain large amounts of information that is irrelevant to DEVS-oriented simulation. Therefore, XMI representations of SysML models, exported from MagicDraw tool, are very large and cumbersome to be directly used for DEVS simulation code generation.

According to DEVSys framework (Figure 1), the next step towards model simulation is the transformation of a SysML PIM described in XMI to a DEVS PIM. In the sense of standardizing the way DEVS models should be specified, independently of how they were constructed (e.g. DEVS SysML models, DEVS visual tools), there is a need to use a DEVS meta-model in terms of a standard meta-modeling facility. Such a facility is MOF, which is provided as a standard by the OMG. Such a meta-model could be used by any standard model transformation tool to define a QVT transformation between SysML and DEVS models represented in XMI, since QVT is a standard set of languages for model transformation from a MOF meta-model to another MOF meta-model.

5.1. DEVS MOF meta-model

Although several DEVS-XML representations are available, none of them is based on a MOF meta-model. After reviewing several DEVS-XML representations discussed in Section 2, we have defined the DEVS MOF meta-model, based on the DEVS-XML version proposed by Risco-Martín et al.[28]

The DEVS MOF meta-model is schematically presented in Figure 10, as a UMLClass Diagram. It contains elements defining structural and behavioral aspects of DEVS atomic (ports, states, internal transition, output, time advance, and external transition functions) and coupled components (ports and coupling). DEVS model structure for both coupled and atomic models is defined in a similar fashion, as proposed by Risco-Martín et al.[28] DEVS model behavior, e.g. DEVS Atomic model functions are described based on system state transitions, also in accordance to DEVS-XML. However, the definition of systems states based on state variables is also included in the proposed meta-model, facilitating the usage of both

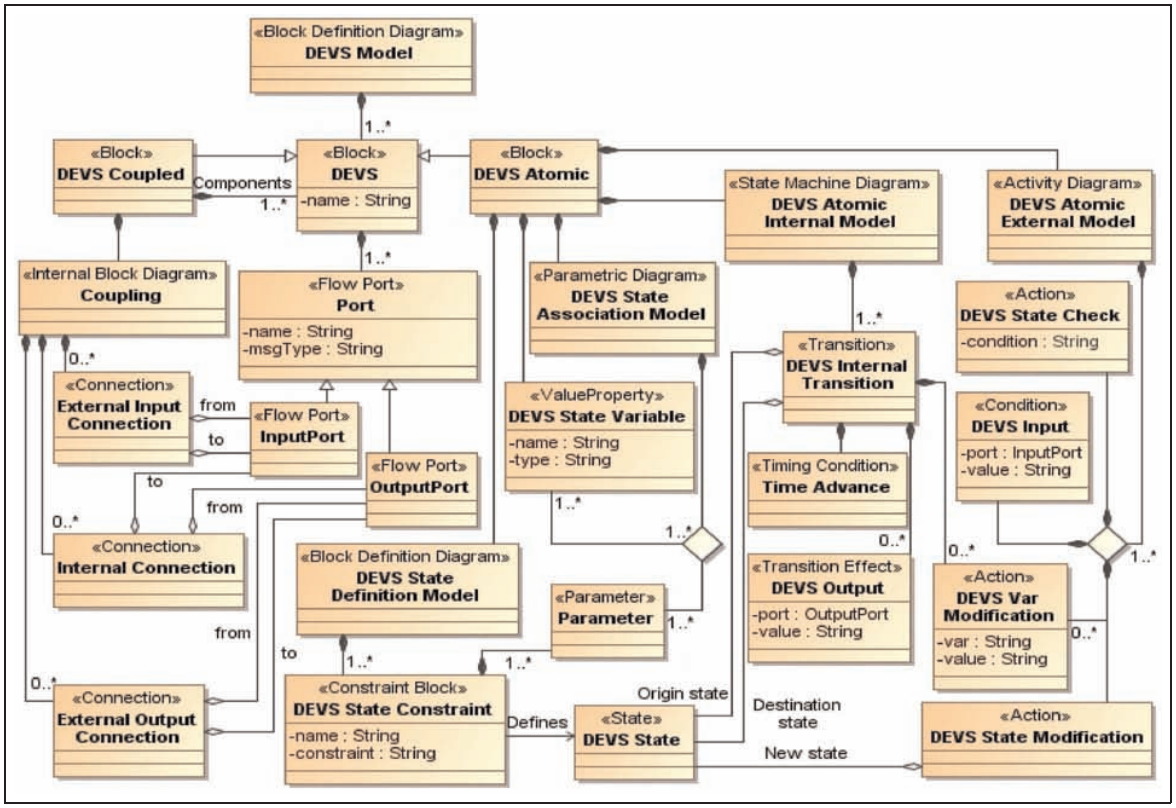

Figure 9. DEVS SysML profile diagram.

state variables and system states to describe behavior functions depending on the simulator used.

Most of the composition associations of the DEVS MOF meta-model class diagram are not named. This implies that the name of the component entity class is used to reference the entity in the context of the composite entity class. This keeps the diagram simpler. However, in cases where the same class is used for two distinct components of an entity (e.g. input and output ports), a distinct name is used for each composition.

It should be noted that the definition of CONDITION and VALUE classes is recursive. For example, a VALUE entity may be composed by one or more other VALUE entities. This is useful and allowed only when an operation is specified, so that a complex expression may be declared (e.g. V1 + V2). In the case of CONDITION, *AND* or *OR* sub-conditions are allowed, to declare complex conditions (e.g. C1 AND C2).

Compared to DEVS-XML,[28] the proposed DEVS MOF meta-model incorporates the relation between state variable values and states. This feature enables the execution of the corresponding DEVS model using a wider variety of DEVS simulators, either implemented in C++ or Java. It also handles complex expression values, utilizing Value class, used to construct state conditions and state variable updates, and provides simpler structure and conceptual coherence. Complex typed values can be supported by serializing them in the String value property, but this aspect has not been extensively exploited so far. These features enhance the meta-model descriptiveness and ease its transformation to executable DEVS code. Therefore, DEVS MOF meta-model presented here establishes a standard, solid foundation for defining DEVS models.

In practice, the proposed DEVS meta-model is defined in terms of MOF elements, so that it can be used within model manipulation tools (i.e. for model transformation), such as the Medini tool.

5.2 Model transformation with QVT

A transformation from UML meta-model, including SysML and DEVS profiles, to the DEVS meta-model is required to convert system models defined in SysML to DEVS models. QVT is an appropriate standard for defining such a transformation, as its main purpose is to define the correspondence and transformation between two meta-models. A set of QVT relations between concepts of DEVS SysML and DEVS MOF meta-models, that

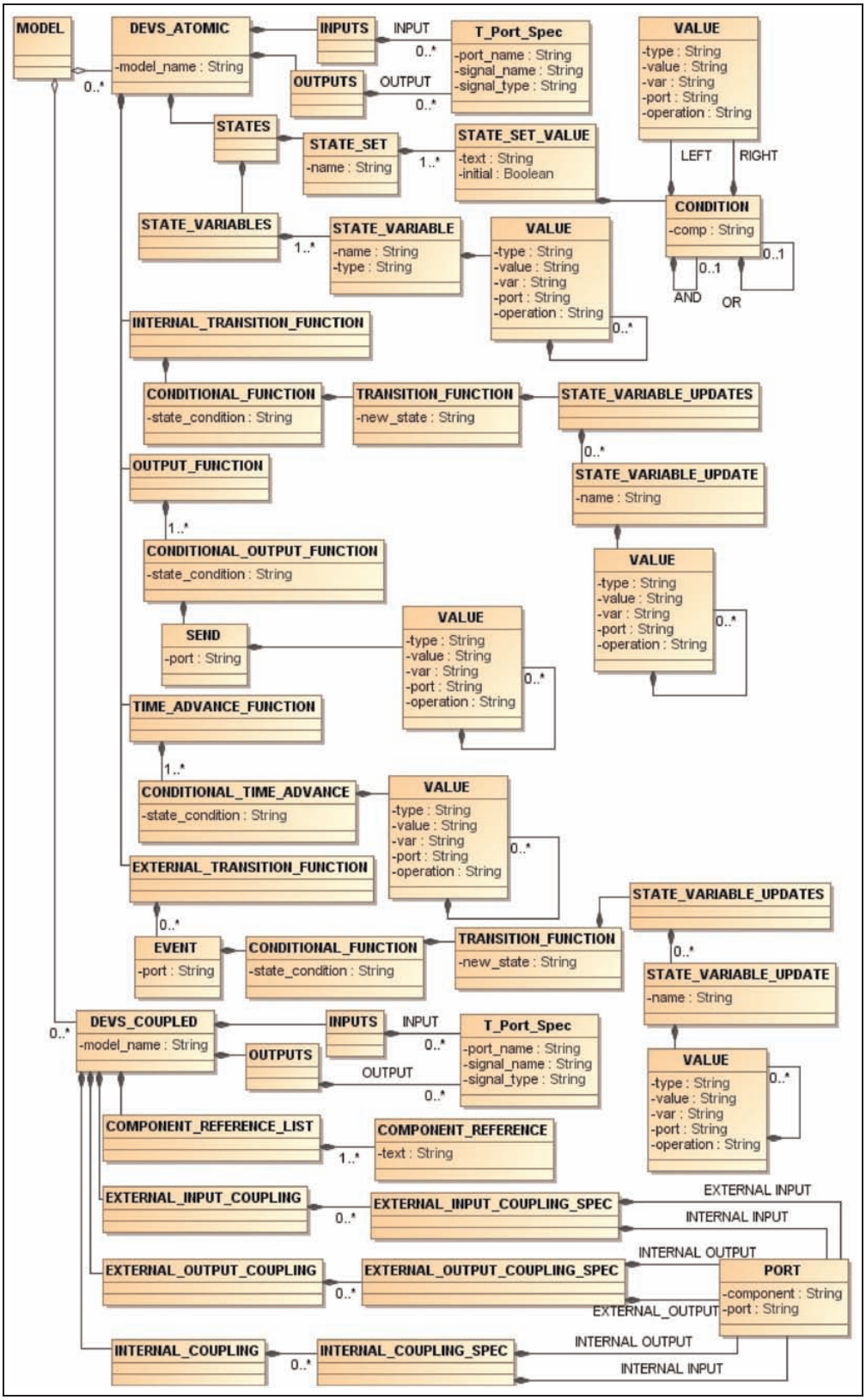

Figure 10. DEVS MOF meta-model.

implement a transformation of models from the first to the second, have been defined. The use of QVT relations for querying the DEVS SysML model facilitates the detection and exploitation of simulation enabled elements, based on their stereotypes and relationships with other elements, regardless of the number and complexity of SysML model diagrams. An Eclipse-based QVT tool (Medini) has been successfully used for the execution of the QVT transformation for various DEVS SysML models.

The transformation from DEVS SysML models (left side) to DEVS models (right side) is outlined in Figure 11, providing a schematic representation of the transformation by indicating the correspondence between parts of the two meta-models. Detailed aspects of both meta-models (presented in Figures 9 and 10) have been omitted to simplify the combined representation. The numbered arrows, denoting several parts of the transformation, are described in the following (no particular order is denoted):

1. *Model transformation*: The topmost part of the transformation. The DEVS SysML model (UML2 entity) contains all DEVS ATOMIC and DEVS Coupled blocks, which are located here, in order to be transformed into DEVS_ATOMIC and DEVS_COUPLED DEVS entities. It is implemented by a QVT relation, which finds all models in the DEVS SysML model (normally there will be only one) and creates a DEVS MODEL for each one. At this point, relations AtomicBlock2DevsAtomic and CoupledBlock2DevsCoupled are applied to each model, so that all DEVS components are identified and transformed.
2. *DEVS common elements transformation*: This part transforms elements that are common in atomic and coupled DEVS models, i.e. model name and input/output ports.
3. *DEVS atomic state definition transformation*: Transforms the DEVS State Constraint blocks that define the state set.
4. *DEVS atomic state variable definition transformation*: Transforms the DEVS State Variable value properties to DEVS State Variables.
5. *DEVS atomic state association transformation*: Transforms the DEVS State Association Model to conditions attached to state set values.
6. *DEVS atomic internal model transformation*: Transforms the DEVS Atomic Internal Model (State Machine Diagram) to Internal Transition Function, Output Function and Time Advance Function.
7. *DEVS atomic external model transformation*: Transforms the DEVS Atomic External Model (Activity Diagram) to External Transition Function.
8. *DEVS coupled components transformation*: Transforms the Component composition associations to the component reference list.
9. *DEVS coupled coupling transformation*: Transforms the port connections of the Internal Block Diagram to External Input Coupling, External Output Coupling and Internal Coupling.

According to Figure 11, DEVS Atomic Internal Model transformation corresponds to arrow 6. *DEVS Atomic Internal Model* defined as a stereotype of a UML2 State Machine diagram is transformed to *INTERNAL_TRANSITION_FUNCTION, OUTPUT_FUNCTION* and *TIME_ADVANCE_FUNCTION* entities of the DEVS meta-model, representing related DEVS functions. Each of them is constructed using information included in the corresponding State Machine diagram. The QVT relations proposed for this part of the transformation are included in Appendix A.

The part of the model that is generated by the SysML-to-DEVS transformation of the Processor DEVS Atomic Internal model (presented in Figure 7), is presented in Code Listing 1, in XMI format.

```
      <NEW_STATE text="idle"/>
      <STATE_VARIABLE_UPDATES>
        <STATE_VARIABLE_UPDATE name="job">
          <VALUE value="Integer(-1)"/>
        </STATE_VARIABLE_UPDATE>
      </STATE_VARIABLE_UPDATES>
    </TRANSITION_FUNCTION>
  </CONDITIONAL_FUNCTION>
</INTERNAL_TRANSITION_FUNCTION>
<OUTPUT_FUNCTION>
  <CONDITIONAL_OUTPUT_FUNCTION>
    <STATE_CONDITION text="busy">
      <SEND port="out">
        <STATE_VARIABLE_VALUE name="job"/>
      </SEND>
    </STATE_CONDITION>
  </CONDITIONAL_OUTPUT_FUNCTION>
</OUTPUT_FUNCTION>
<TIME_ADVANCE_FUNCTION>
  <CONDITIONAL_TIME_ADVANCE>
    <STATE_CONDITION text="busy"/>
    <TIME_ADVANCE>
      <VALUE type="Real" value="3.5"/>
    </TIME_ADVANCE>
  </CONDITIONAL_TIME_ADVANCE>
  <CONDITIONAL_TIME_ADVANCE>
    <STATE_CONDITION text="idle"/>
    <TIME_ADVANCE>
      <VALUE type="Real" value="Infinity"/>
    </TIME_ADVANCE>
  </CONDITIONAL_TIME_ADVANCE>
</TIME_ADVANCE_FUNCTION>
```

Code Listing 1. Processor DEVS atomic internal model functions in XMI (according to DEVS MOF meta-model).

Transitions are used to define all functions. *CONDITIONAL_FUNCTION* elements are described by the *origin state*, while *TRANSITION_FUNCTIONS* corresponding to a CONDITIONAL_FUNCTION are described by the *destination state*. In the case of

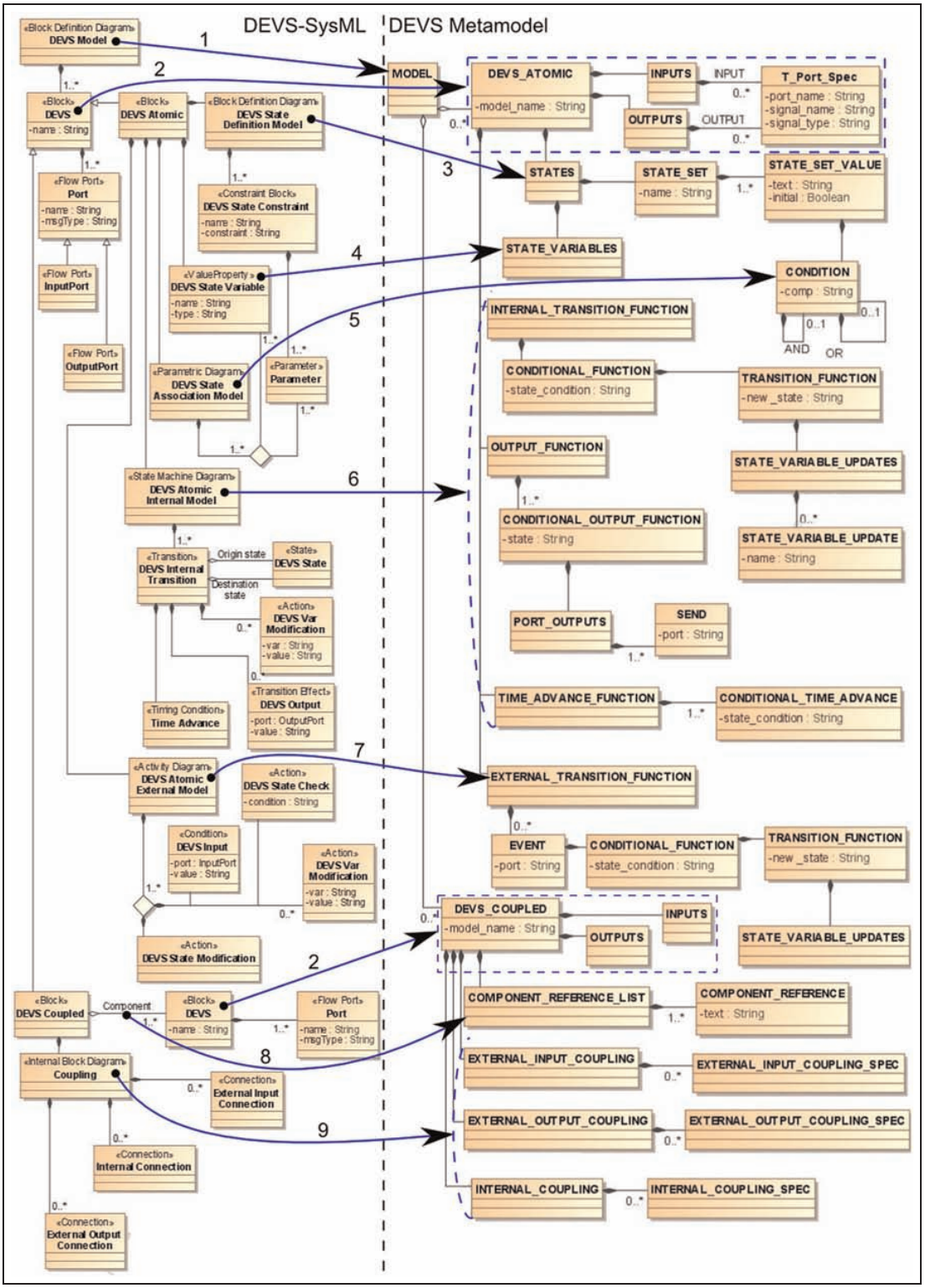

Figure 11. DEVS SysML to DEVS transformation.

INTERNAL_TRANSITION_FUNCTION element, a *STATE_VARIABLE_UPDATES* element is also added to indicate state variable modification, as indicated by *DEVS Var Modification Action* associated to the *Transition*. In the case of Processor system (Figure 7) the state variable Job, of type integer, is decreased by 1.

6. DEVS Code Generation and Execution

According to the DEVSys framework (Figure 1) and given that a DEVS PIM is generated, as described in previous sections, its transformation to an executable DEVS PSM is the last step of the process. This is feasible, since all necessary information for the DEVS PSM construction is included in the DEVS PIM. To generate a platform-specific DEVS representation, a specific DEVS simulation environment must be selected, so that the proper transformation from DEVS models to the appropriate target format (XML, Java code, etc.) may be defined. The XLSC DEVS platform was selected as the simulation environment,[26] mainly due to the fact that it supports an XML representation of the DEVS model to be simulated. The DEVS PIM model (in XMI format) is syntactically transformed to XLSC XML by an XSLT transformation. Since the two formats have rather simple syntactical differences, the XLSC XML file is constructed with the help of a set of XSLT templates that match DEVS XMI elements, from which the required values are retrieved and placed in the XLSC elements. Figure 12 illustrates DEVS meta-model, the general structure of XLSC XML, and the main steps of the transformation between them.

The defined XSLT transformations are rather simple. Details on them are provided in Appendix B. Generally, DEVS model elements are transformed to the respective XLSC elements. However, one should note that:

- In XLSC, there is no definition of the state set. Only state variables are defined.
- In XLSC, all DEVS atomic functions (internal transition, output, time advance and external transition) are defined in a low level, procedural manner. Thus, they are not further analyzed in the figure. In our XSLT transformation, we basically build the procedural XLSC elements that implement the behavior declared in the respective DEVS model elements.
- Coupling is represented in a simpler and flat way in XLSC. Couplings are not distinguished as internal or external (input and output). For each coupling, a source (component, port) and a target (component, port) are specified. When the source or target component is a DEVS coupled model, then the component attribute is set to ''this''. Otherwise, the name of the (component) DEVS model is used.

As an example, the XLSC equivalent of the processor output function (Code Listing 1) is presented in Code Listing 2. It is generated as the result of DEVS-to-XLSC transformation on the DEVS XMI representation of the output function.

```
<outputFunction>
  <action>
    <if>
      <condition>
           <equal>
                 <retrieve state="phase"/>
                 <string>busy</string>
           </equal>
      </condition>
      <then>
           <output port="out" variable="item_nr">
                 <retrieve state="job"/>
           </output>
           <send message="Item" port="out"/>
      </then>
    </if>
  </action>
</outputFunction>
```

Code Listing 2. Processor output function in XLSC XML.

The generated XLSC code is passed to the XLSC interpretation environment that dynamically creates DEVSJava classes for the atomic and coupled DEVS models. The simulation model can be executed in DEVSJava simulation environments. Figure 13 shows the execution of the EFP model in the SimView DEVSJava component.

As reported by Kapos et al.,[40] a varied version of the framework that supports existing simulation library components has been tested with more complex scenarios of Enterprise Information System configurations.

7. Conclusions and future work

Generating simulation code in an automated fashion for system models defined in SysML, may enhance system design validation. In this paper, an integrated framework called DEVSys was discussed that enables the transformation of system models defined in SysML to DEVS executable models in a fully automated manner. To do so, the properties of SysML system models are enriched with simulation specific capabilities, in order to generate executable simulation models, based on the DEVS formalism. The proposed framework was tested to provide automated generation of DEVS executable code for the XLSC DEVS simulator. Each step was defined, implemented and applied as well. Moreover, MDA concepts have been widely adopted, rendering DEVSys an open, standards-based and extensible framework. Hence, it may be further applied using a variety of UML modeling tools and/or DEVS simulation environments.

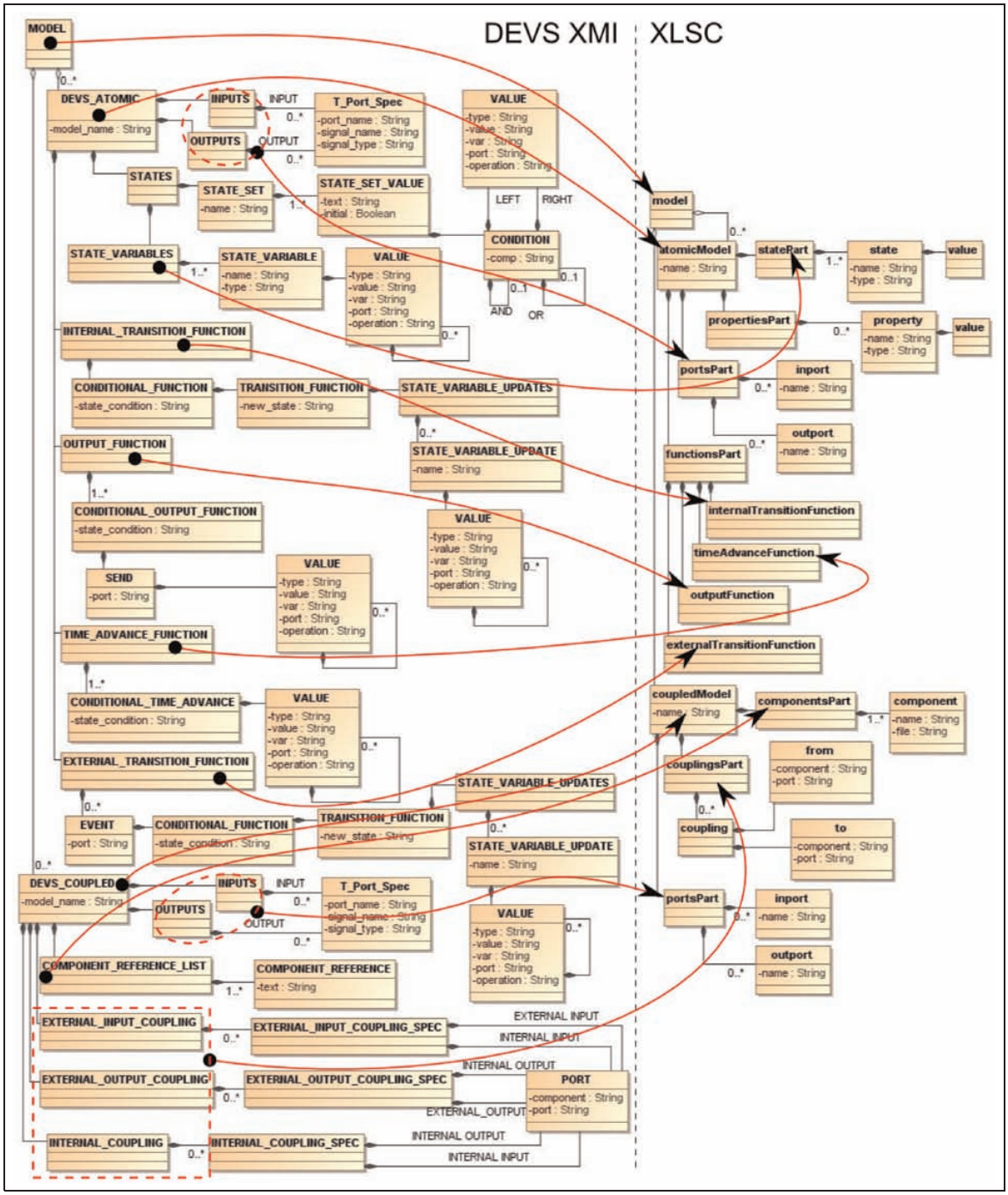

Figure 12. DEVS to XLSC transformation.

QVT with OCL were successfully employed to transform SysML models to DEVS models represented in XMI, based on the DEVS MOF meta-model. This meta-model enables the transformation of DEVS models to executable DEVS XLSC simulation code and, also, serves as the basis for creation of DEVS PIMs that can be used by any standard model transformation tool supporting QVT. This way, generation of DEVS simulation models from system models defined in any standard modeling language is enabled. A simple example demonstrating the current version of the framework's implementation can be found at http://galaxy.hua.gr/~gdkapos/DEVS_SysML_en.htm.

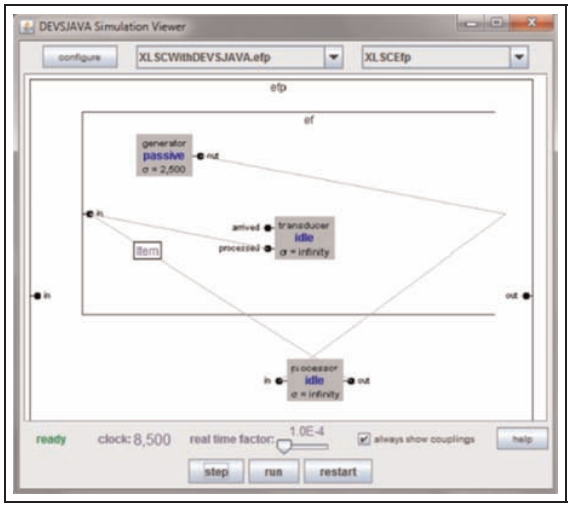

Figure 13. EFP simulation execution in DEVSJava.

Future work involves (a) applying the proposed framework and related tools in a variety of real-world cases, (b) combining the capability of the framework to define behavioral aspects with selective use of existing, executable simulation components, when they are available, (c) integrating additional DEVS simulators in the framework, and (d) providing additional capabilities within the DEVS profile to enable the system engineer to integrate simulation results within the SysML system model.

Acknowledgements

The authors would like to thank Nicolas Meseth, Patrick Kirchhof, and Thomas Witte for their valuable help.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

AQ1

References

1. Haskins C. INCOSE systems engineering handbook: a guide for system life cycle processes and activities, version 3. INCOSE-TP-2003-002-03, 2006.
2. Law AM. *Simulation modeling and analysis*. 4th ed.McGraw-Hill Series in Industrial Engineering and Management Science. McGraw-Hill, 2006.
3. Object Management Group (OMG). Systems Modeling Language (SysML) specification, version 1.3. www.omg.org/spec/SysML/1.3/PDF (2012).
4. Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. www.omg.org/spec/UML/2.4.1/Superstructure/PDF/ (2011).
5. Sarjoughian HS and Elamvazhuthi V. CoSMoS: a visual environment for component-based modeling, experimental design, and simulation. In: *Proceedings of the 2nd international conference on simulation tools and techniques for communications, networks and systems, SimuTools 2009*, Rome, Italy, 2009, p.59.
6. Lee EA, Hylands C, Janneck J, et al. *Overview of the Ptolemy project*. EECS Department, University of California, Berkeley, CA, 2001. Report no. UCB/ERL M01/11. www.eecs.berkeley.edu/Pubs/TechRpts/2001/3947.html.
7. Lee EA. Ptolemy project vision. *Presented at the 8th Biennial Ptolemy Miniconference*. http://chess.eecs.berkeley.edu/pubs/550.html (2009).
8. Mathworks. Matlab and Simulink. www.mathworks.com (2011).
9. Zeigler BP, Praehofer H and Kim T. *Theory of modeling and simulation*. 2nd ed.Academic Press, 2000.
10. McGinnis L and Ustun V. A simple example of SysML-driven simulation. In: *Proceedings of the 2009 winter simulation conference*, Austin, TX, pp.1703–1710. IEEE, 2009..
11. Schonherr O and Rose O. First steps towards a general SysML model for discrete processes in production systems. In: *Proceedings of the 2009 winter simulation conference*, Austin, TX, pp.1711–1718. IEEE, 2009.
12. Huang E, Ramamurthy R and McGinnis LF. System and simulation modeling using SysML. In: *WSC'07: proceedings of the 39th winter simulation conference*, Piscataway, NJ, pp.796–803. IEEE Press, 2007.
13. Peak RS, Burkhart RM, Friedenthal SA, et al. Simulation-based design using SysML. Part 1: A parametrics primer. In: *INCOSE international symposium*, San Diego, CA, 2007, pp.1–20.
14. Peak R, Paredis CJJ and Tamburini DR. *The composable object (COB) knowledge representation: enabling advanced collaborative engineering environments (CEEs), COB requirements and objectives (v1.0)*. Atlanta, GA: Georgia Institute of Technology, 2005.
15. Tamburini DR. Defining executable design and simulation models using SysML. www.pslm.gatech.edu/topics/sysml/ (2006).
16. Paredis CJJ and Johnson T. Using OMG's SysML to support simulation. In: *WSC'08: proceedings of the 40th winter simulation conference*,. 2008, pp.2350–2352.
17. Wang R and Dagli CH. An executable system architecture approach to discrete events system modeling using SysML in conjunction with colored Petri Nets. In: *IEEE systems conference 2008*, Montreal, pp.1–8. IEEE Computer Press, 2008.
18. Schamai W. Modelica modeling language (ModelicaML): a UML profile for Modelica. http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-20553 (2009).
19. Kerzhner AA, Jobe JM and Paredis CJJ. A formal framework for capturing knowledge to transform structural models into analysis models. *J Simul* 2011; 5: 202–216.
20. Object Management Group (OMG). SysML–Modelica transformation (SyM). www.omg.org/spec/SyM/1.0/PDF/ (2012).
21. Object Management Group (OMG). Model driven architecture. Version 1.0.1. www.omg.org/cgi-bin/doc?omg/03-06-01.pdf (2003).
22. Nikolaidou M, Dalakas V, Mitsi L, et al. A SysML profile for classical DEVS simulators. In: *Proceedings of the third international conference on software engineering advances (ICSEA 2008)*, Malta, pp.445–450. IEEE Computer Society, 2008.

23. Zeigler BP and Sarjoughian HS. Introduction to DEVS modeling and simulation with Java. DEVSJava Manual. www.acims.arizona.edu/PUBLICATIONS/publications.shtml (2003).
24. Wainer GA and Giambiasi N. Timed Cell-DEVS: modelling and simulation of cell spaces. In: Sarjoughian H and Cellier F (eds) *Title?*. Springer-Verlag, 2001.
25. Zhang M, Zeigler BP and Hammonds P. DEVS/RMI – an auto-adaptive and reconfigurable distributed simulation environment for engineering studies. *Int Test Eval Assoc J* 2005; 27: 49–60.
26. Meseth N, Kirchhof P and Witte T. XML-based DEVS modeling and interpretation. In: *SpringSim '09: proceedings of the 2009 Spring Simulation Multiconference*, San Diego, CA, pp.1–9. Society for Computer Simulation International, 2009.
27. Mittal S, Risco-Martín JL and Zeigler BP. DEVS/SOA: a cross-platform framework for net-centric modeling and simulation in DEVS unified process. *Simulation* 2009; 85: 419–450.
28. Risco-Martín JL, Mittal S, López-Peña MA, et al. A W3C XML schema for DEVS scenarios. In: *SpringSim '07: proceedings of the 2007 spring simulation multiconference*, San Diego, CA, pp.279–286. Society for Computer Simulation International, 2007.
29. Risco-Martín JL, De La, Cruz JM, Mittal S, et al. eUDEVS: Executable UML with DEVS theory of modeling and simulation. *Simulation* 2009; 85: 750–777.
30. Object Management Group (OMG). MOF 2 XMI mapping specification, Version 2.4.1. www.omg.org/spec/XMI/2.4.1/PDF/ (2013).
31. Object Management Group (OMG). Meta object facility (MOF) core specification, Version 2.4.1. www.omg.org/spec/MOF/2.4.1/PDF/ (2013).
32. Object Management Group (OMG). Meta object facility (MOF) 2.0 query/view/ transformation specification, Version 1.1. www.omg.org/spec/QVT/1.1/PDF/ (2011).
33. MG. *SysML plugin for Magic Draw*. Magic Draw, 2007.
34. Batarseh O and McGinnis LF. System modeling in SysML and system analysis in Arena. In: *Proceedings of the 2012 winter simulation conference (WSC'12)*, pp. 258:1–258:12. http://dl.acm.org/citation.cfm?id=2429759.2430107.
35. Hosking M and Sahin F. An XML-based system of systems discrete event simulation communications framework. In: *SpringSim'09: Proceedings of the 2009 spring simulation multiconference*, 2009, pp.1–9. San Diego, CA: Society for Computer Simulation International.
36. Mittal S, Risco-Martín JL and Zeigler BP. DEVSML: automating DEVS execution over SOA towards transparent simulators. In: *DEVS symposium, spring simulation multiconference*, 2007, pp.287–295. ACIMS Publications.
37. Hwang MH and Zeigler BP. Reachability graph of finite and deterministic DEVS networks. *IEEE Trans Autom Sci Eng* 2009; 6: 468–478.
38. Object Management Group (OMG). OMG object constraint language (OCL), Version 2.3.1. www.omg.org/spec/OCL/2.3.1/PDF/ (2012).
39. World Wide Web Consortium (W3C). Extensible stylesheet language transformations (XSLT). www.w3.org/TR/xslt20 (2007).
40. Kapos GD, Dalakas V, Tsadimas A, et al. Model-based system engineering using SysML: deriving executable simulation models with QVT. In: *IEEE systems conference 2014*, pp.531–538. IEEE Interactive Electronic Library (IEL), IEEE Xplore, 2014.

Author biographies

George-Dimitrios Kapos is currently performing research for his PhD thesis on automated validation of system models via simulation at the Department of Informatics and Telematics, Harokopio University of Athens, Greece. In parallel, he works as an analyst and software developer at the IT Department of the Greek Consignment Deposit & Loans Fund, also participating in efforts for realization of e-Government in Greece. He obtained his BSc in Informatics and an MSc degree in Advanced Information Systems, both with honors from Informatics & Telecommunications Department, National Kapodistrian University of Athens, Greece. His research interests include model-based systems validation, simulation, distributed-object systems dynamic behavior, and distributed, heterogeneous databases homogenization.

Vassilis Dalakas obtained a BSc in Physics, a MSc degree (honors) with specialization in digital signal processing, and a PhD degree with specialization in digital communications, all from the University of Athens (UoA), Greece, in 1998, 2002, and 2010, respectively. Since 2001, he has been affiliated with the Harokopio University of Athens (HUA), Greece, as a Research Fellow (2001–2007 in the Department of Geography and since 2008 in the Department of Informatics and Telematics) and as a network and system administrator since 2005. His research interests include wireless digital communications for satellite and terrestrial system applications, digital signal processing techniques, as well as modeling and simulation standardization methods. In these areas, he has co-authored several papers, two book chapters and was a co-recipient of the 2006 Best Paper Award in Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE).

Mara Nikolaidou is a Professor in the Department of Informatics and Telematics at Harokopio University of Athens. She holds a PhD and Bachelor degree on Computer Science from Department of Informatics and Telecommunications at University of Athens. Her research interests include software and information system engineering, service-oriented architectures, e-government, and digital libraries. Over the last years she actively participated in numerous projects on service-oriented architectures, digital libraries and e-government. She has published more than 100 papers in international journals and conferences.

Dimosthenis Anagnostopoulos is a Professor in the Department of Informatics and Telematics at Harokopio University of Athens. He holds a PhD and Bachelor degree on Computer Science from Department of Informatics and Telecommunications at University of Athens. He has published more than 100 papers in international journals and conferences. His research interests include discrete event simulation, faster-than-real-time simulation, modeling and simulation of distributed information systems. He has actively participated in numerous projects related to simulation, e-government and information systems.

Appendix. Meta-model definitions and transformations

A. Transforming SysML to DEVS models with QVT

The transformation from SysML to DEVS models is an important element of the proposed framework. In order to provide an implementation perspective, the part of the QVT transformation that identifies SysML block state transitions and creates the corresponding DEVS component behavioural characteristics is presented in Code Listing 3.

```
transformation devsSysml2devsMM(devsSysml:uml, devs:Devs) {
    top relation SysmlModel2DevsModel {
            checkonly domain devsSysml ms : uml::Model {};
            enforce domain devs mx : Devs::MODEL {};
            where {
                    CoupledBlock2DevsCoupled(ms,mx);
                    AtomicBlock2DevsAtomic(ms,mx);
            }
    }
    ...
    relation Transitions2ConditionalFunctions {
            ns, nt : String;
            b : Sequence(String);
            checkonly domain devsSysml r : uml::Region {
                    transition = tr : uml::Transition {
                            source = s : uml::Vertex { name = ns },
                            target = t : uml::Vertex { name = nt },
                            effect = e : uml::FunctionBehavior { _body = b }
                    }
            };
            enforce domain devs itf : Devs::T_Internal_Transition_Function {
                    CONDITIONAL_FUNCTION = cf : Devs::T_Conditional_Function {
                            STATE_CONDITION = sc : Devs::PCDATA { text = ns },
                            TRANSITION_FUNCTION = tf : Devs::T_Transition_Function {
                                    NEW_STATE = nst : Devs::PCDATA { text = nt },
                                    STATE_VARIABLE_UPDATES =
                                            svus : Devs::T_State_Variable_Updates {}
                            }
                    }
            };
            when {
                    not (tr.source.name = '');
                    tr.getAppliedStereotype('DEVS_SysML::DEVS State Transition')->notEmpty();
            }
            where {
                    b->first().split('cmd:')->select(x|x.startsWith('stateVarUpdate'))->
                            forAll(y|StringSequence2StateVarUpdate(y,svus));
            }
    }
    relation StringSequence2StateVarUpdate {
            checkonly domain devsSysml b : String { };
            enforce domain devs svus : Devs::T_State_Variable_Updates {
                    STATE_VARIABLE_UPDATE = svu : Devs::T_State_Variable_Update {
                            name = b.substring(b.indexOf(' ')+2,b.indexOf(',')),
                            VALUE = v : Devs::T_Value {}
                    }
```

```
        };
        where {
                value(b.substring(b.indexOf(',')+2,b.lastIndexOf(')')),v);
        }
}
relation Transitions2ConditionalOutputFunctions {
        ns : String;
        b : Sequence(String);
        checkonly domain devsSysml r : uml::Region {
                transition = tr : uml::Transition {
                        source = s : uml::Vertex { name = ns },
                        effect = e : uml::FunctionBehavior { _body = b }
                }
        };
        enforce domain devs of : Devs::T_Output_Function {
                CONDITIONAL_OUTPUT_FUNCTION = cof : Devs::T_Conditional_Output_Function {
                        state = ns,
                        PORT_OUTPUTS = po : Devs::T_Port_Outputs {}
                }
        };
        when {
                tr.getAppliedStereotype('DEVS_SysML::DEVS State Transition')->notEmpty();
                b->first().indexOf('cmd:output') > -1;
        }
        where {
                b->first().split('cmd:')->select(x|x.startsWith('output'))->
                        forAll(y|StringSequence2PortOutputs(y,po));
        }
}
relation StringSequence2PortOutputs {
        checkonly domain devsSysml b : String { };
        enforce domain devs po : Devs::T_Port_Outputs {
                SEND = s : Devs::T_Send {
                        port = b.substring(b.indexOf('(')+2,b.indexOf(',')),
                        VALUE = v : Devs::T_Value {}
                }
        };
        where { value(b.substring(b.indexOf(',')+2,b.lastIndexOf(')')),v); }
}
relation Transitions2ConditionalTimeAdvances {
        ns, tav : String;
        checkonly domain devsSysml r : uml::Region {
                transition = tr : uml::Transition {
                        source = s : uml::Vertex { name = ns },
                        guard = g : uml::Constraint {
                                specification = sp : uml::DurationInterval {
                                        min = m : uml::Duration {
                                                expr = e : uml::LiteralString { value = tav }
                                        }
                                }
                        }
                }
        };
        enforce domain devs taf : Devs::T_Time_Advance_Function {
                CONDITIONAL_TIME_ADVANCE = cta : Devs::T_Conditional_Time_Advance {
                        STATE_CONDITION = sc : Devs::PCDATA { text = ns },
                        TIME_ADVANCE = ta : Devs::T_Time_Advance {
                                VALUE = v : Devs::T_Value {}
                        }
                }
        };
```

```
            when { tr.getAppliedStereotype('DEVS_SysML::DEVS State Transition')->notEmpty(); }
            where { value(tav,v); }
    }
    relation NoTransitions2InfiniteConditionalTimeAdvances {
            ns : String;
            checkonly domain devsSysml r : uml::Region {
                    subvertex = sv : uml::Vertex { name = ns }
            };
            enforce domain devs taf : Devs::T_Time_Advance_Function {
                    CONDITIONAL_TIME_ADVANCE = cta : Devs::T_Conditional_Time_Advance {
                            STATE_CONDITION = sc : Devs::PCDATA { text = ns },
                            TIME_ADVANCE = ta : Devs::T_Time_Advance {
                                    VALUE = ev : Devs::T_Value {
                                            type = 'Real',
                                            value = 'Infinity'
                                    }
                            }
                    }
            };
            when {
                    sv.getAppliedStereotype('DEVS_SysML::DEVS State')->notEmpty();
                    sv.getOutgoings()->isEmpty();
            }
    }
}
```

Code Listing 3. DEVS conditional functions transformation with QVT.

In relational QVT, a transformation between two domains is defined through a set of relations that map entities of one domain to entities of the other. The transformation is initiated by the application of *top relations*, which usually reference other relations. The two domains (*devsSysml* and *devs*) are declared and characterized. *devsSysml* is marked as *checkonly*, since it is the source domain and does not need to be updated to satisfy conditions declared in the relation. On the other hand, domain *devs* is characterized as *enforce*, as this is the target domain that needs to be constructed according to the conditions stated in the relation. The relation must be enforced in this case. Regarding the transformation of state transitions of SysML blocks' state machine diagrams, six relations are used. *Transitions2ConditionalFunctions* creates DEVS Atomic Internal Transition Function, while *StringSequence2StateVarUpdate* handles the appropriate state variable modification specification. *Transitions2 ConditionalOutputFunctions* defines DEVS Atomic Output Function with the aid of *StringSequence2PortOutputs. Transitions2ConditionalTimeAdvances* defines DEVS Atomic Time Advance function for states that will be current for finite amounts of time, while *NoTransitions2 InfiniteConditionalTimeAdvances* identifies states without outgoing transitions.

Transitions2ConditionalOutputFunctions is described in more detail in the following. Precisely, this relation identifies *DEVS Internal Transitions*, their *Origin states* and *DEVS Outputs*, and maps them to the corresponding *CONDITIONAL_OUTPUT_FUNCTION*, *SEND* and *VALUE* elements. For each domain, the elements of interest are declared: *transition* with its *source* and *effect* for *devsSysml*, and *CONDITIONAL_OUTPUT_FUNCTION* with its *STATE_CONDITION* for *devsXml*. Variables are used to match values between the domains, e.g. *ns* variable is used to match *transition.source.name* with *CONDITIONAL_OUTPUT_FUNCTION.STATE_CONDITION.text*. The *when* clause of a relation defines the preconditions that must be met for this relation to be applied. The clause may contain other relation invocations or OCL expressions. In this example, *tr* (the transition) must have the stereotype *DEVS State Transition*. This means that a *CONDITIONAL_OUTPUT_FUNCTION* element will be created only for each transition that is stereotyped as *DEVS State Transition*. The *where* clause of a relation defines the expressions (possibly with other relations) that should be applied after this relation is applied and the domain elements are created. The relation *StringSequence2Output* parses the output command *y* and creates the appropriate elements under the *sc* state condition element. In this case, an OCL expression results in multiple invocations of a relation. The first element of the string sequence *b* is splinted in all its substrings that are delimited by the 'cmd:' string, resulting is a list of all commands. From those, only the ones starting with the string 'output' (only output commands) are selected.

Medini tool has been used for both the definition of the DEVS meta-model and the definition and execution of the QVT transformation from SysML models to DEVS models. In Figure 14, the Medini GUI for creating and viewing MOF meta-models is illustrated.

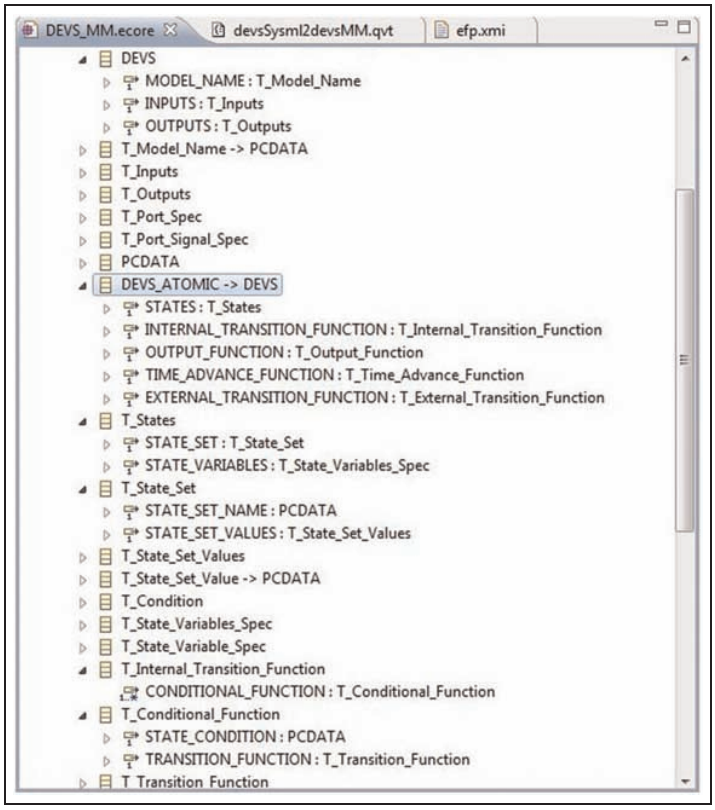

Figure 14. DEVS MOF meta-model definition within Medini tool in terms of MOF elements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="2.0"
    exclude-result-prefixes="xs xdt err fn xmi DevsXml"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:fn="http://www.w3.org/2005/xpath-functions"
    xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
    xmlns:err="http://www.w3.org/2005/xqt-errors"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:DevsXml="urn:DevsXml.ecore">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/xmi:XMI">
  <model>
      <xsl:apply-templates/>
  </model>
  </xsl:template>
  ...
  <xsl:template match="INTERNAL_TRANSITION_FUNCTION">
    <internalTransitionFunction>
      <action>
        <xsl:for-each select="CONDITIONAL_FUNCTION">
          <if>
            <condition>
              <equal>
                <retrieve state="phase"/>
                <string>
                  <xsl:value-of select="STATE_CONDITION/@text"/>
                </string>
              </equal>
            </condition>
          <then>
              <update state="phase">
```

```
            <string>
              <xsl:value-of select="TRANSITION_FUNCTION/NEW_STATE/@text"/>
            </string>
          </update>
          <xsl:for-each
               select="TRANSITION_FUNCTION/STATE_VARIABLE_UPDATES/STATE_VARIABLE_UPDATE">
            <update>
              <xsl:attribute name="state">
                <xsl:value-of select="@name"/>
              </xsl:attribute>
              <xsl:call-template name="expression"/>
            </update>
          </xsl:for-each>
       </then>
       </if>
     </xsl:for-each>
     <update state="phase_changed">
       <integer>1</integer>
     </update>
  </action>
  </internalTransitionFunction>
</xsl:template>
<xsl:template match="TIME_ADVANCE_FUNCTION">
  <timeAdvanceFunction>
    <action>
      <if>
        <condition>
          <equal>
            <retrieve state="phase_changed"/>
            <integer>1</integer>
          </equal>
        </condition>
        <then>
          <xsl:for-each select="CONDITIONAL_TIME_ADVANCE">
            <if>
              <condition>
                <equal>
                  <retrieve state="phase"/>
                  <string>
                    <xsl:value-of select="STATE_CONDITION/@text"/>
                  </string>
                </equal>
              </condition>
              <then>
                  <update state="sigma">
                    <xsl:for-each select="TIME_ADVANCE">
                      <xsl:call-template name="expression"/>
                    </xsl:for-each>
                  </update>
              </then>
            </if>
          </xsl:for-each>
          <update state="phase_changed">
            <integer>0</integer>
          </update>
        </then>
      </if>
    </action>
  </timeAdvanceFunction>
</xsl:template>
<xsl:template match="OUTPUT_FUNCTION">
```

```
    <outputFunction>
      <action>
        <xsl:for-each select="CONDITIONAL_OUTPUT_FUNCTION">
          <if>
            <condition>
              <equal>
                <retrieve state="phase"/>
                <string>
                  <xsl:value-of select="@state"/>
                </string>
              </equal>
            </condition>
            <then>
              <xsl:for-each select="PORT_OUTPUTS/SEND">
                <output>
                  <xsl:attribute name="port">
                    <xsl:value-of select="@port"/>
                  </xsl:attribute>
                  <xsl:attribute name="variable">v_<xsl:value-of select="@port"/>
                  </xsl:attribute>
                  <xsl:call-template name="expression"/>
                </output>
                <send message="Item">
                  <xsl:attribute name="port"><xsl:value-of select="@port"/></xsl:attribute>
                </send>
              </xsl:for-each>
            </then>
          </if>
        </xsl:for-each>
      </action>
    </outputFunction>
  </xsl:template>
</xsl:stylesheet>
```

Code Listing 4. DEVS internal, time advance, and output functions transformation with XSLT.

Figure 15 illustrates the QVT editor of the Medini tool. As shown in the left part of the figure, DEVS_MM.ecore is the file containing the definition of DEVS MOF meta-model. The UML2 meta-model is integrated in the tool. Medini allows relational QVT transformation execution and debugging.

B. Creating XLSC executable models from DEVS models with XSLT

In order to be able to execute DEVS models in XMI format, they should be transformed to an executable format (XLSC documents in this case). XSLT -the de facto standard for syntactic transformations of XML documents with different formats- was used for this purpose. In the same context as in Appendix A, the XSLT templates transforming DEVS Internal Transition, Time Advance and Output Functions to the respective XLSC elements are listed in Code Listing 4.

The XSLT transformation was defined using EditX tool that provides XSLT execution capabilities and is freely available. A screenshot of the tool is provided in Figure 16. It depicts the simple transformation template that matches *STATE_VARIABLES* elements of DEVS models to *StatePart* elements of XLSC XML documents.

As shown in Figure 16, the defined *xsl* elements, control how the transformation is performed, while simple elements are the skeleton of the result of the transformation, which is formed and enriched by the xsl command elements. For example, the first element (*xsl:template*) indicates that all commands contained in it should be executed for every STATE_VARIABLES element found in the context of this template. The second element (*xsl:for-each*) indicates that all the commands contained in it should be executed for every STATE_VARIABLE element contained exactly under the VARIABLES element found in the previous line. The state element that follows will be added to the output for each STATE_VARIABLE element. The following *xsl:attribute* commands attach two attributes (name and type) at the state element. The value of the first one (name) will be the same as the value of the name attribute of the source STATE_VARIABLE element. The value of the other attribute (type) emerges by

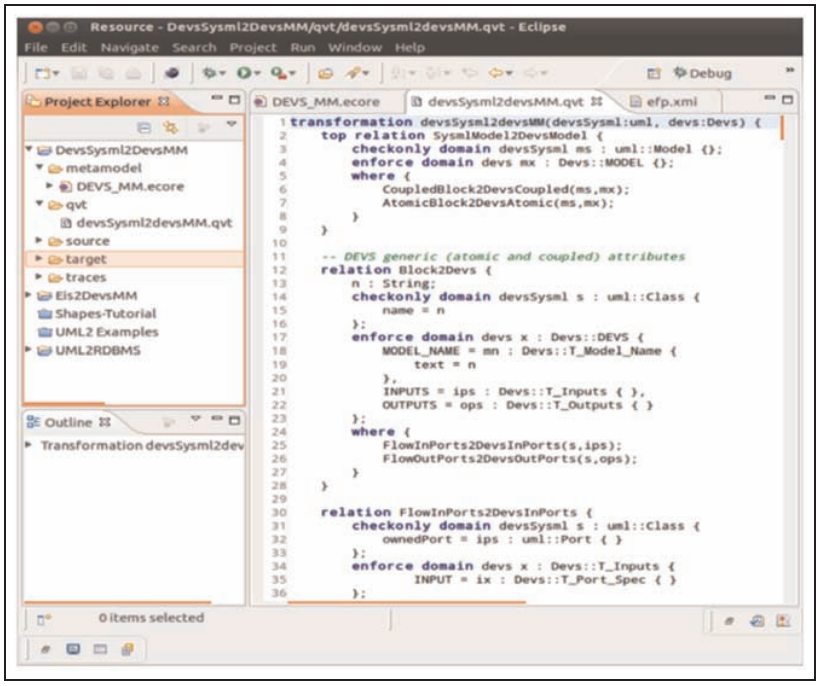

Figure 15. UML2 to DEVS MOF transformation with Medini.

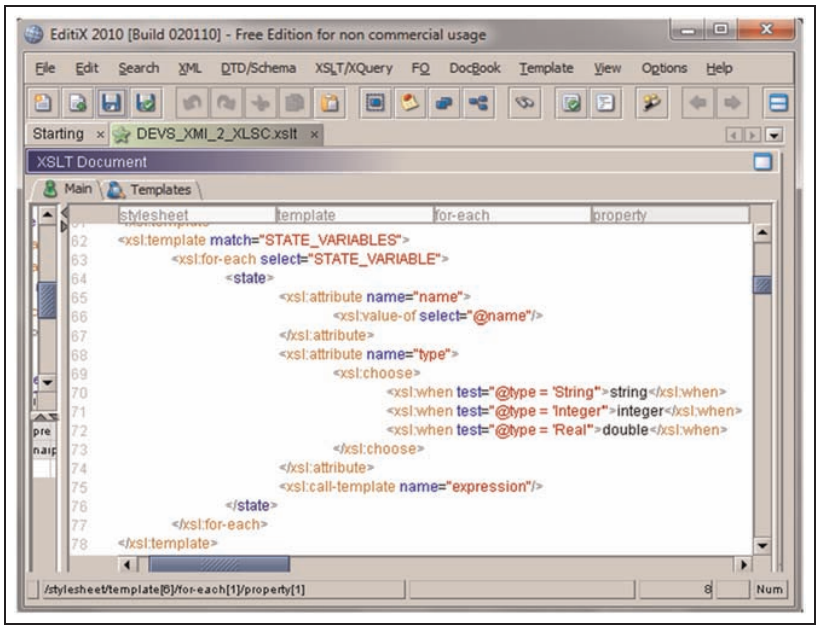

Figure 16. DEVS XMI to XLSC transformation with EditX.

properly transforming the value of the type attribute of the source STATE_VARIABLE element, as indicated by the *xsl:choose* and *xsl:when* elements. Finally, the expression template is invoked (*xsl:call-template*), so as to find the initial value of the STATE_VARIABLE element and transform it to the appropriate state value.