XML-based DEVS modelling and simulation tracking

Youcef Dahmani*

Laboratory of Energy Engineering and Computer Engineering,
University Ibn Khaldoun Tiaret, Algeria
Email: dahmani_y@yahoo.fr
*Corresponding author

Hemza Nedjari Benhadj Ali and Abdelkader Boubekeur

Department of Computer Science,
University Ibn Khaldoun Tiaret, Algeria
Email: nedjari.ba.hemza@gmail.com
Email: boubekeur.kader@gmail.com

Abstract: Discrete event system specification (DEVS) is a modelling and simulation formalism for discrete event dynamic systems. DEVS has known different variants and software implementations to meet specific needs. Unfortunately, these implementations do not necessarily share models and cannot capitalise on model reuse and data exchange. XML, as a meta-language, seems to be an appropriate language for helping modellers with weak programming knowledge. Indeed, XML is adopted as a standard for information exchange and knows an increasingly important use on the web. This work aims at determining an XML-based implementation of DEVS for devising an efficient modelling and simulation framework. After defining and validating our XML schemas, an execution step of these models is defined based on our XML abstract simulator which executes a set of XSLT transformations and generates an XML simulation tree describing the different steps of the system during the simulation time.

Keywords: discrete event system specification; DEVS; XML schema definition; XSD; extensible stylesheet language transformations; XSLT; simulation tree; modelling; simulation.

Reference to this paper should be made as follows: Dahmani, Y., Ali, H.N.B. and Boubekeur, A. (2020) ‘XML-based DEVS modelling and simulation tracking’, *Int. J. Simulation and Process Modelling*, Vol. 15, Nos. 1/2, pp.155–169.

Biographical notes: Youcef Dahmani is a Professor in the Department of Computer Science at University Ibn Khaldoun Tiaret. He earned his Engineer degree in 1992 from U.S.T. Oran, Algeria, and MSc degree in 1997 from University of Es Senia Oran and obtained PhD in 2006 from U.S.T. Oran. His research interests include network security, modelling and simulation, artificial intelligence, information systems, fuzzy logic and reactive robotic systems.

Hemza Nedjari Benhadj Ali is a Software Engineer graduated from University Ibn Khaldoun Tiaret, Algeria. His work focuses specifically on software comprehension and developing, the idea behind this work is to provide tools helping to automatise the implementation and test of large scale softwares. His best hobby is organising events of scientific popularisation.

Abdelkader Boubekeur holds a Master’s degree in Software Engineering from University Ibn Khaldoun Tiaret, Algeria. He works as a System and Network Administrator and his best research topic is modelling and simulation.

1 Introduction

Nowadays, DEVS is an appropriate means of using modelling and simulation in design phases of any dynamic system. Because of its generality in systems specification, the DEVS formalism has been successfully applied to several application in different programming environments such as ADEVS, CD++, DEVS-Suite, PowerDEVS, PythonDEVS and Ururau (Peixoto et al., 2017; Tendeloo and Vangheluwe, 2017; Hollmann et al., 2015).

There are several implementations for expressing DEVS models. Modellers have to learn the programming language of each simulator apart. They have also another obstacle in sharing models from these different implementations. That is, in order to model a system of a particular domain with DEVS, in addition to the domain specialist who describes the model, a computer scientist must be ubiquitous to translate the model into object code.

Using a metalanguage may help in reducing this drawback. Indeed, for a modeller who does not need a lot of programming knowledge, XML seems to be an appropriate solution for enabling automatic transformations from XML representation into given simulation software. In fact, XML offers several advantages and is increasingly being adopted as a standard for information interchange.

In recent years, many researchers have focused their research topics on XML applied to many domain and subjects. The importance of using XML is still increasing in different domains such as modelling and simulation (Mittal and Risco-Martín, 2017b; Mittal and Douglass, 2012; Wainer and Mosterman, 2010).

In the present work, an XML vocabulary of DEVS formalism is formally defined by an XML schema against which the vocabulary can be validated. To keep track of different simulations, we have also designed an XML abstract simulator. The simulation is done via XSLT transformations generating an XML simulation tree at each event occurrence, which would allow tracing the simulation of DEVS model. The idea emerges from the fact that at each event, the model is in a state that could be represented by a tree, and this tree evolves over the simulation time.

The rest of this paper is described as follows. The next section (Section 2) provides an overview of related works to tackle the problem of DEVS models and simulators interoperability by conceptual modelling language. Some works on XML applied to DEVS are also presented. Section 3 recalls some technical background, naturally we focus on the DEVS modelling and simulation formalism. In Section 4, we illustrate our proposed XML schema of DEVS atomic and coupled model. Section 5 describes the different XSLT transformation rules to generate our simulation tree in compliance with the DEVS formalism. A case study of this work is detailed in Section 6. Section 7 focuses on some performance evaluation of our proposed simulator and the size of files generated by the DEVS models and their simulators through our automatic XSLT transformations rules. In Section 8, a conclusion and discussion on the present work are given and some future works.

2 Related work

The DEVS formalism is one of the common formalism used in the simulation of dynamic systems (Zeigler, 1976; Zeigler et al., 2000; Wainer and Mosterman, 2010). There is a large number of software implementations of this formalism, involving the creation of many groups that are not able to share models and cannot capitalise on model reuse (Martín et al., 2007; Meseth, 2011; Garredu, 2013; Hollmann et al., 2015).

Many works were proposed by describing a formal modelling language which is able to model and simulate DEVS models without requiring programming concepts. In Hong and Kim (2006), the authors have defined DEVSpecL which is an abstract description of DEVS models. In fact, it is a specification language for modelling and simulation of discrete event systems. However this language supports only basic features of this formalism and needs some user's API to build complex types. The work of Seo (2009) defines a DEVS namespace to implement an interoperable DEVS simulation environment using the service-oriented architecture (SOA). To ensure interoperability between DEVS simulators, the author proposes a new design of DEVS simulator service based on three layers, a protocol layer which provides basic functionality to simulate DEVS models, message layer provides type information to DEVS simulator and report layer which provides the result of the simulation. The procedure of creating the DEVS simulator is based on Java resulting in DEVS simulator service. To interoperate with other software implementations, a specific DEVS simulator service is required. Wang and Lu (2002) have proposed an XML-based DEVS modelling tool to improve the interoperability of simulations. They based their work on a higher abstraction modelling framework called system entity structure (SES). It is a structured knowledge representation scheme that is applied to guide the synthesis of models from the base of the model. However, this metamodel requires a higher level of abstraction, which is not necessarily mastered by all modellers. Another approach in Garredu (2013) has been proposed to tackle the problem of DEVS models and simulators interoperability. The author proposes a metamodel named MetaDEVS based on Model Driven Engineering process to realise his work. A focus is done on models interoperability rather than simulators interoperability. To make these models productive, the framework of Eclipse EMF including its metamodel Ecore was retained as the environment of metamodelling and code generation. Transformation rules from a model into another model are implemented using the Atlas transformation language (ATL). The transformation from a model into a text applied to DEVS in order to generate code compatible with PythonDEVS using its templates is implemented with Eclipse Acceleo plugin. Object constraint language (OCL) is used to have a control over the relationships between classes and specification of constraints independently of simulation platforms. However, this work focuses on creating a new metamodel instead of using existing metamodels and improving them. On the other hand, to create the simulation model, the author defines some templates or code patterns to generate the code for specific platforms. To use it effectively, the modeller must have an idea of how they work. The challenge of this work is the lack of a standardised representation of DEVS models despite some works of Simulation Interoperability

Standards Organisation and DEVS Standardization Group (Garredu, 2013). In Bazoun et al. (2016), ATL and XSLT are also used to obtain final DEVS models that can be viewed and simulated by the DEVS simulator. In fact, ATL transformation is applied to transform conceptual level of service processes models into business process modelling and notation (BPMN). Then these models are transformed into DEVS models by XSLT to simulate the behaviour of the entire process model in order to see its evolution to validate and verify the desired dynamic behaviour of these models in their future real environment. However, this method is well adapted for business oriented people and no description of software implementation for the DEVS editor is described.

The work of Hollmann et al. (2015) proposes CML-DEVS, a conceptual modelling language for DEVS which expresses DEVS models in terms of logical and mathematical expressions such as loop, type, and functions. However, if one wishes to generate code for a new Modelling and Simulation software library, he needs to add a new code generator. This code is based on translation rules specific to the programming language at issue. In Casas (2015), the author uses the graphical specification and description language (SDL) to transform a DEVS model into this language. Algorithms are given to do these transformations and the modeller has to know this language to use it effectively. The work in Zeigler et al. (2017) uses the DEVS natural language (DNL). The behavioural model can be built from a graphic user interface which is converted into a DNL file which in turn is transformed into a Java file for simulation. Yet, this solution needs expertise on Java and Finite probabilistic DEVS (FP-DEVS) which is one of many extensions of DEVS formalism. The DNL file is made of atomic model component, data component, and Java component and the simulation software named MS4 Me runs in an Eclipse environment.

However, with the growing use of the internet, these works cannot be automatically parsed, analysed and translated into a specific programming language of simulation modelling. It is noticeable in the literature that a code based on extensible markup language (XML) may be a well suited candidate (Meseth, 2011). With its increasing use in different areas of sciences and research, XML has become a standard format to store and exchange information over the internet. It offers several advantages like the validation of schema, restriction of data types, transformation into a specific format, etc. (Bray et al., 2006).

This relevance has advantages for using XML technology to support DEVS systems. Indeed, storing DEVS models in XML format offers more information exchange and a convenient way to specify translations of XML in web browser document (Wainer and Mosterman, 2010). Thus, the idea of writing DEVS formalism with XML without any support from a specific programming language and validating the DEVS XML schema is needed.

An early work was introduced by Fishwick (2002) followed by Janoušek et al. (2006) proposing XML representations for DEVS models, using JavaML; however for the coupled model, a graphical tool was proposed. Another work was proposed by Martín et al. (2007) to specify the DEVS formalism in W3C XML schema. This work lacks processing instruction such as loops and conditions; in fact, just the data aspect of the models is presented. Mittal et al. (2007) and Mittal (2007) propose DEVSML, in their work, they suggest modifications from the DEVS formalism into a DEVS service to meet the service-oriented architecture model. However, the DEVSML description in XML is validated by the standardised document type definition (DTD) which provides a very weak specification language compared to XML schema which is more powerful and flexible and meets today's programming needs. However, a revised version was proposed in DEVSML 2.0 (Mittal and Douglass, 2012) where they describe a grammar and a domain specific language for this formalism. A recent work was also developed to deploy formal discrete event dynamic system simulation based on DEVSML and SOA architecture in a cloud environment (Mittal and Risco-Martín, 2017a).

On the other hand, there are several XML solutions that are mainly intended for the variants of DEVS. The goal is to derive these different variants from the XML file without specific programming knowledge. DEVSimPy (Santucci and Capocchi, 2014) is an implementation that deals with fuzzy DEVS, DEVS-XML is another implementation for the variant FD-DEVS. (**?**), Works of Meseth (2011) focus on an XML interpreter called XLSC, the author developed an XML-based language in form of interface definition language (IDL) interfaces of the Object Management Group Inc (OMG). It is designed for the parallel-DEVS formalism, it expresses DEVS models and their behaviours in an XML-based manner. An XLSC interpreter for Java was developed from the abstract IDL interfaces to simulate XLSC models with the DEVS simulator DEVSJAVA framework. The idea behind these works is to prove that XML-based simulation language can be interpreted instead of compiled or translated.

The solutions are multiple, expertise and an adaptation for each software are needed. Therefore, a metalanguage like XML is pertinent in this area for ensuring the life cycle of a modelling and simulation process of a system. Using XML in the modelling and simulation phases offers an advantage where all results are in text format and therefore understandable by any modeller. All information on DEVS models will be provided, all details on models and simulation results over the time would be formulated exclusively in text document format. Consequently, an alternative for these works is required to fully express a DEVS model, its static and behavioural aspects, in textual form.

3 Background

3.1 DEVS modelling

The DEVS formalism was developed by Zeigler (1976). It is based on the mathematical theory of dynamic systems and known as one of the common formalism used in the simulation of systems which temporal and spatial behaviours are complex to be treated analytically. It is known for its ability for coupling heterogeneous models and separating the modelling process from the simulation process. DEVS formalism allows two levels of description. At the lowest level, a basic part called DEVS atomic model which describes the autonomous behaviour of a discrete event system. At the highest level, a DEVS coupled model describes a system as a modular and hierarchical structure (Zeigler, 1976; Zeigler et al., 2000).

3.1.1 DEVS atomic model

A DEVS atomic model is based on continuous time, inputs, outputs, states and four characteristic functions (output, internal, external, time advance). It describes the functional aspect of the system. Formally, a DEVS atomic model is described by a seven-tuple:

$$MA = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (1)$$

where

- X the set of input events, each input being characterised by a pair (port number/value)
- S the set of states
- Y the set of output events, each output being characterised by a pair (port number/value)
- $\delta_{int} : S \rightarrow S$ internal transition function, represents the states changes caused when the elapsed time reaches the lifetime of the state
- $\delta_{ext} : Q \times X \rightarrow S$ external transition function, defines how an input event changes a state of the system where $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ total states and e describes the elapsed time since the last transition of the current state s
- $\lambda : S \rightarrow Y$ when the elapsed time reaches the lifetime of the state, this function generates an output event
- $ta : S \rightarrow \mathbb{R}^+$ time advance function, it is the lifetime of a state.

3.1.2 DEVS coupled model

The DEVS coupled model enables a modular and hierarchical framework. It allows the creation of complex models starting from atomic and/or coupled models, formalising the modelled system through a set of inter-connected and reused components. A DEVS coupled model is defined as an eight-tuple:

$$\begin{aligned} MC = \langle X, Y, D, \{M_d \mid d \in D\}, \\ EIC, EOC, IC, Select \rangle \end{aligned} \quad (2)$$

- X set of inputs events
- Y set of outputs events
- D is the name set of sub-components
- $M_d \mid d \in D$ set of sub-components, which are either DEVS atomic or coupled model
- EIC: set of external input coupling
- EOC: set of external output coupling
- IC: defines the internal coupling;
- $Select$: $2^D \rightarrow D$: tie-break selector which defines priority between concurrent events.

3.2 DEVS simulation

3.2.1 Abstract DEVS simulation

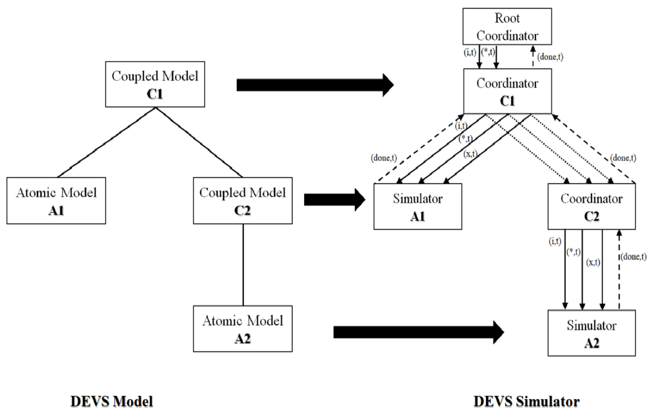
Zeigler et al. (2000) has developed the concept of abstract simulator. Bypassing any programming language and any simulation platform, the specification of a simulation is obtained by matching each element of the model with its corresponding component of the simulator (Figure 1). The first interest of this hierarchical representation of the simulation structure is based on the possibility of transforming directly a hierarchical model into an executable simulation code. Another advantage of this representation is its support of the interoperability of formalisms. To perform a simulation, a hierarchy of processors, equivalent to the model hierarchy, is built. This simulation structure exploits the hierarchical and modular aspects resulting from the representation of DEVS models. There are three types of processors:

- $Simulator$ assigned to each atomic model, ensures the simulation of atomic models using DEVS characteristic functions $(\delta_{int}, \delta_{ext}, \lambda, ta)$.
- $Coordinator$ assigned to each coupled model, ensures routing of messages between coupled models according to the definitions of coupling.
- *Root-coordinator* ensures the overall management of the simulation. It orders the beginning and the end of the simulation and manages the global time.

3.2.2 Abstract simulator operation

The operation of an abstract simulator involves handling four types of messages: $Xmessage$: carries external input information; $Ymessage$: carries external output information; $*message$: indicates an internal event is due; Imessage: initiates simulation, initial conditions of time and state must first be set in all simulators of the hierarchy. In the literature, there is another message which may be either implicit or explicit; it is used for scheduling (synchronising) the abstract simulators. Indeed after each executed message, an acknowledgement is sent back in form of $donemessage$ containing the next and the last time event.

Figure 1 Mapping a hierarchical model into a hierarchical simulator



The DEVS formalism gives a specification of a system, for which initial values of the states are defined. From these data, it is necessary to specify how to generate the following states and the output trajectories. To do this, simulation algorithms are defined in an abstract simulator which implements them.

When a $Simulator$ receives $*message$, it calls the model output function λ to generate an output event $Ymessage$ and sends it to its parent coordinator based on the old state s. Then, it computes the new state by means of the internal transition function. When an atomic simulator receives an $Xmessage$, it executes the external transition function of its associated atomic model. After processing an $Xmessage$ or $*message$, the $Simulator$ sends an acknowledgement message including the last and next time event to its parent coordinator to prepare a new schedule. In an atomic $Simulator$, the last time event t_l as well as the local state s are kept.

The $Coordinator$ is responsible for synchronising exchange of messages between the simulators. When an $Imessage$ isred received, the coordinator forwards it to all its direct children, and gets the next event times and last event times back, and saves them in an event list. When a coordinator receives $*message$, it selects an imminent component (atomic/coupled) by means of the tie-breaking function $Select$ specified in the coupled model and routes the message to the chosen component. After the execution of the subcomponent, the coordinator, updates the list of events and informs its parent. When a coordinator receives a $Xmessage$ from its parent coordinator, it routes the message into the affected component by checking which components are connected to the port from which this event comes. After that, it updates its last and next time event and sends them to its parent. When a coordinator receives $Ymessage$ from the imminent submodel, it consults the external coupling relationships defining the output ports influenced by this model to know if it must transmit the message to its parent. Then it looks at the internal coupling relationships to determine the successor and ports that may be influenced by this event. In this case, it transforms the $Ymessage$ into $Xmessage$ for destination to the influential successors, and then updates its event list to send the last and next time event.

The $root\text{-}coordinator$ is basically a time scheduler, its algorithm contains the global simulation loop. At the beginning of each simulation run, the $root-coordinator$ sends $Imessage$ to its child (attached simulator/coordinator). The messages that initiate a simulation cycle are the $*message$. The $root\text{-}coordinator$ transmits these messages to its successors which respond by sending back the date of the next time event t_n. Then, it repeatedly sends $*message$ to its children and computes the top next time event until the simulation ends. The simulation cycle continues as long as there is $*message$ to be processed, or as long as a stopping condition is not satisfied (events occurred, end of simulation, etc.).

4 XML-DEVS modelling

4.1 DEVS atomic model schema definition

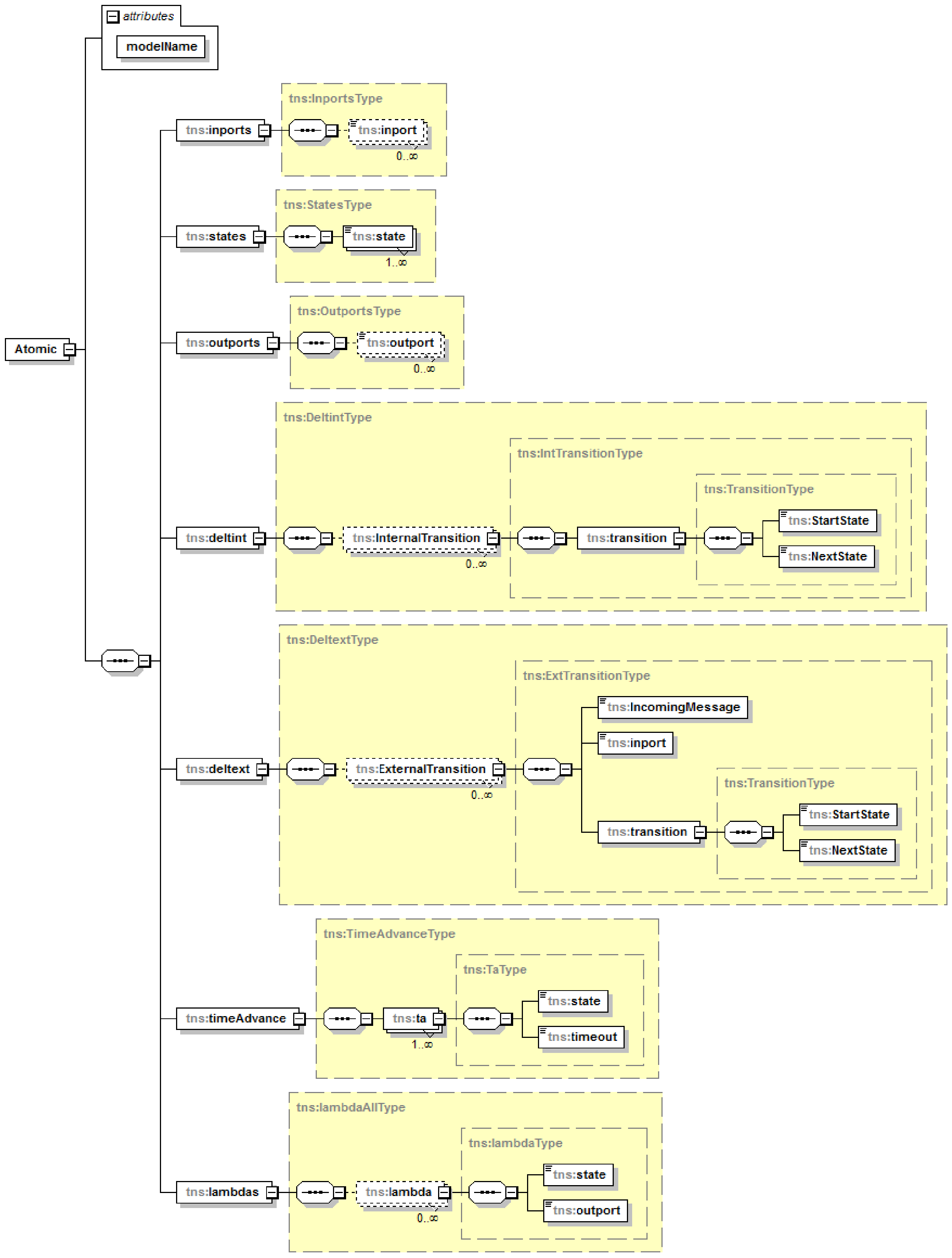
Considering a large number of DEVS software implementations, our specification of the DEVS formalism by XML schema is mainly based on DEVSML (Meseth, 2011; Mittal and Douglass, 2012; Mittal and Risco-Martín, 2017b; Hollmann et al., 2015) allowing the generation of models using all DEVS concepts without depending on a specific simulation platform. These schemas are based on the classic DEVS paradigm.

In its basic form, the schema of the DEVS Atomic model consists of seven elements and one attribute: the top level element is $Atomic$ (Figure 2). It states that an atomic model, besides the attribute $modelName$ which is supposed unique for each one, the model has a sequence of child elements, in which each of the seven parts may appear exactly once.

Here is the description of child elements for each particular part. In this work, we have kept the same structure as defined in Mittal and Douglass (2012).

- $inports$ element: set of input port elements.
- $states$ element: consists of set of sequence of $state$ element. Each state element is assigned a string as a value.
- $outports$ element: set of output port elements.
- $deltint$ element: a complex type which defines the functions of the internal transition δ_{int}. Each transition is characterised by $StartState$ and $NextState$ element. An internal transition represents the change from starting state to the next one;.
- $deltext$ element: this element represents the set of external transitions δ_{ext}.

Figure 2 DEVS atomic model schema view (see online version for colours)



Each external transition is defined by:

a $IncomingMessage$ element: the message value indicating the triggering of the transition

b $inport$ element: the associated port to the external transition

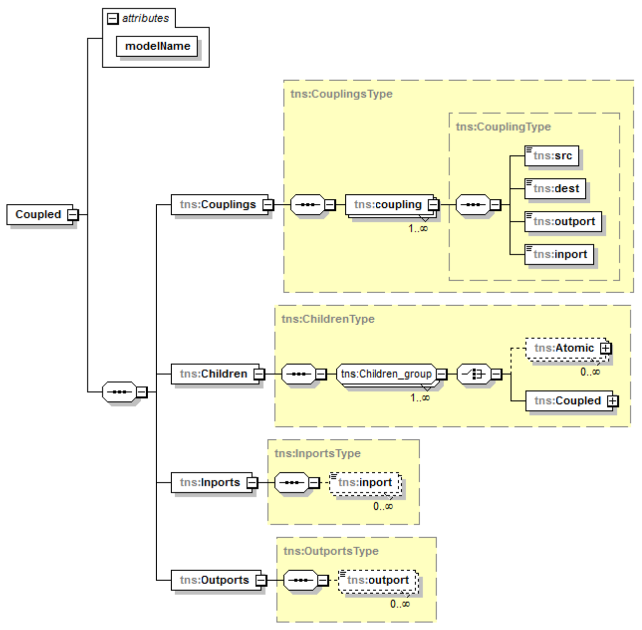
c $transition$: consists of $StartState$ and $NextState$ element representing the transition from a state to another.

- $TimeAdvance$ element: set of ta elements which assign for each state a lifetime (timeout).
- $lambdas$: the set of $lambda$ elements, affects for each state an output port $outport$ to send an external message.

4.2 DEVS coupled model schema definition

The coupled model is composed of atomic or/and coupled models. In the present work, the schema of the coupled model is composed of the classical elements as defined in Mittal and Douglass (2012). The difference between their works lies in the addition of element $Children$ which respects the class hierarchy of the DEVS scheme defined in Luh (1994) and Zeigler et al. (2000).

Figure 3 DEVS coupled model schema view (see online version for colours)



In XML schema of DEVS coupled model (Figure 3), the top level element is $Coupled$ and composed of one attribute and four elements:

- $modelName$: the attribute model name.
- $Couplings$: this element represents the different couplings IC, EIC, OIC. A coupling is defined by the couple (model, port). The model may be either a source or a destination $(src, dest)$ element. The port may be either input or output port $(inport, outport)$ element.
- $Children$: set of schemas of its children.
- $Inports$: represents all input ports of the model.
- $Outports$: set of output ports of the model.

In the present work, when simultaneous events happen, the last message in the list is processed. Execution order follows the order of the definition of schemas in the hierarchy of the model.

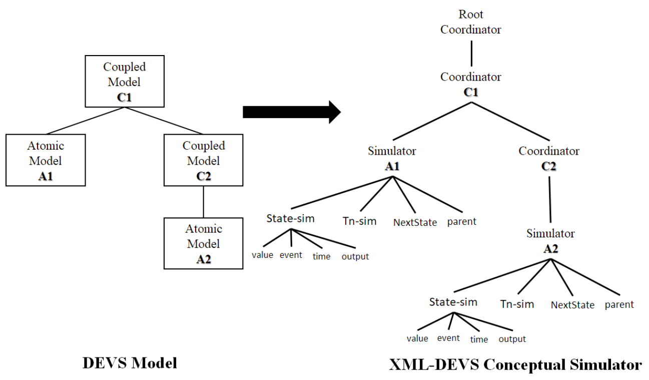
All graphical representations of XML schema Definition in this work are created with XMLSpy software of Altova. This tool was useful to validate our model and for sake of comprehension, visual schemas have been presented in this paper.

5 XML-DEVS simulation tracking

5.1 XML-DEVS conceptual simulator

The aim of the DEVS simulation is to reproduce the behaviour of the system under particular experimental conditions and observe its evolution over time; the entity capable of reproducing this behaviour is the simulator. Considering the simulation structure, our abstract simulator consists of three types of processors: $root - coordinator$, $coordinator$ and $simulator$ as in Zeigler et al. (2000). All these processors adhere to a message exchange protocol that coordinates the simulation tracking. This exchange can be designed and developed as a simulation tree based on XSLT transformations. This tree (Figure 4) represents our system at any time of the simulation which is inspired by the class hierarchy of DEVS scheme, for which we add a new part at the bottom of the structure describing the different states occurred during the simulation time (trace of states, event type, date of event occurrence, ...)

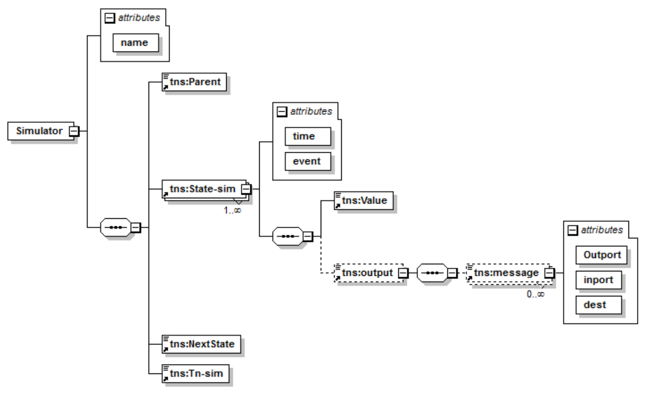
Figure 4 Mapping model onto simulation tree



5.1.1 XML-DEVS simulator

Basically, the dynamic behaviour of the DEVS model is performed by the component $Simulator$ associated with the atomic model; when receiving events, it activates the functions defined in the model. The schema of our simulator (Figure 5) defines all aspects expressing the state of this model over time. A simulator consists of:

Figure 5 The schema view of simulator



The top element $Simulator$ is composed of:

- $name$: this attribute is the atomic model name;
- $Parent$: This element is the name of the parent coordinator;
- $NextState$: represents the next state once internal transition processed;
- $Tn\text{-}sim$: This element contains the date of the next event to be processed;
- $State\text{-}sim$: this element is inserted in the tree during the simulation.

The element $State\text{-}sim$ is composed of two attributes and two elements:

- $time$: this attribute indicates the time when the event was inserted in the simulator tree.
- $event$: this attribute gives the type of event that generated the insertion of this state in the tree. In other words, the cause that led the simulator to add this state. There are three possible types of events:
  - a $Init$: initialisation event, represents the receipt of $Imessage$
  - b $Internal$: event triggered by an internal transition, represents receipt of $*message$
  - c $External$: event triggered by an external transition, represents receipt of $Xmessage$.
- $Value$: state value.
- $output$: this element represents the result of the execution of the function λ called through the internal transition δ_{int}. Each output element is composed of a message containing a string indicating the value of the message and three attributes described as below:
  - a $Outport$: port by which the message will be sent
  - b $inport$: input port of the destination receiving the message
  - c $dest$: destination $Simulator$ component.

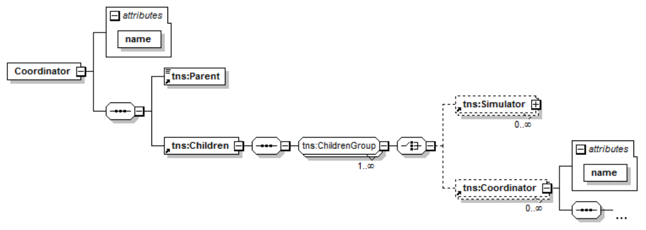
By the end of the simulation, each simulator will contain the different states by which it has passed through.

5.1.2 XML-DEVS coordinator

The $Coordinator$ component is responsible for the synchronised exchange of messages between $Simulator$, it represents coupled models. In this work, the schema of a $Coordinator$ (Figure 6) consists of two elements and one attribute:

- $name$: this attribute is the name of the coupled model;
- $Parent$: name of the parent coordinator;
- $Children$: definition of child coordinator/simulator.

Figure 6 Coordinator schema view

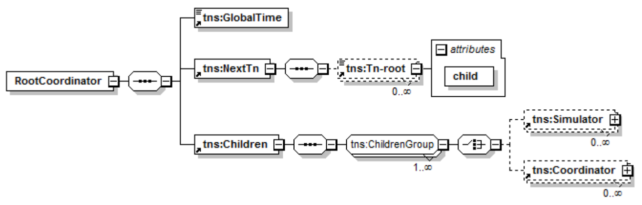


5.1.3 XML-DEVS root-coordinator

The $Root\text{-}coordinator$ holds the global simulation time, it manages simulation with $*message$. The subordinate of $Root\text{-}coordinator$ sends out these messages to its successors. The schema of $Root\text{-}coordinator$ is represented by a diagram (Figure 7) consisting of three elements:

- $GlobalTime$: global time of simulation.
- $NextTn$: list of elements $Tn\text{-}root$. Each of these elements contains a value representing the minimum value of the next expected internal transition. This value applies to one or more $Simulator$ components at a time. The $Tn\text{-}root$ element includes the attribute $child$ corresponding to the name of $Simulator$ that contains this minimum value. $Tn\text{-}root$ is the minimum value of all $Tn\text{-}sim$ in the tree.
- $Children$: definition of $Coordinator/Simulator$ which is the most top model in the tree.

Figure 7 Root coordinator schema view



The processing of the various messages by the *Simulator* components is done via XSLT transformations.

5.2 Transformations XSLT and simulation tracking

The different structures already presented are part of the simulation tree, which represents the trace of the system change over time. These changes are saved in the XML file *SimulationTree.xml*.

Simulation tree undergoes a series of transformations representing the behaviour of the models. To simulate the behaviour of these models, we have defined transformation files which will run in the following steps.

5.2.1 Data processing

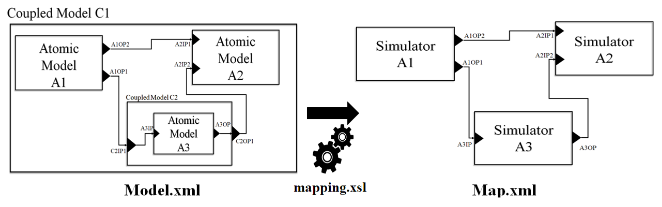
The model of the system created by the user is the file *Model.xml*, once created, it is loaded in order to generate the simulation tree file. The first step is to browse the model structure in order to establish the links between the different submodels. The result of this mapping step is a file *Map.xml*.

5.2.1.1 Mapping

Basically, coupled models represent the hierarchy of a model; the behavioural aspect is assumed by the atomic models. Therefore, we are interested in the *Simulator* rather than the *Coordinator* which only ensures the synchronisation between the submodels.

As the transitions functions are treated by the *Simulators*, this part of mapping will extract all the atomic models and the links between them while disregarding the couplings. We obtain a flat representation of the links (Figure 8). This phase will exclusively allow us to deal with the *Simulators* and the map file will result in links between all atomic models of the whole model.

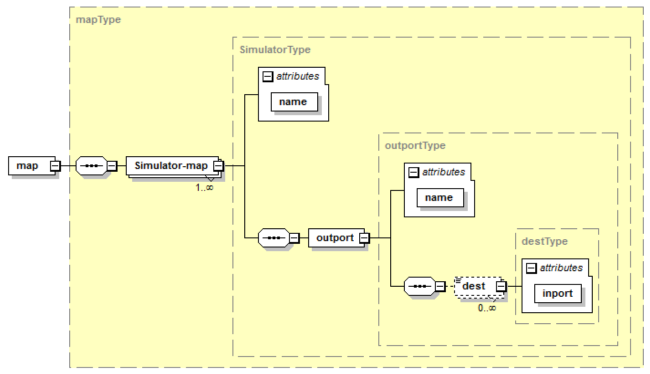
Figure 8 Mapping models onto flat architecture



A flat mapping of our model will generate a map file described by the schema of Figure 9. It depicts a link between two atomic models, a source and a destination model. On the other hand, we specify the output and the input port respectively of the source model and destination model.

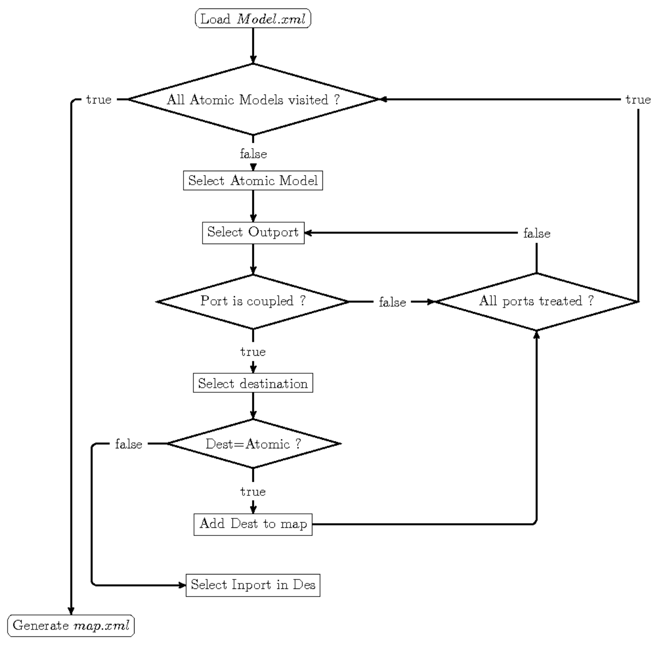
Hence, each *Simulator-map* element consists of a set of elements *outport* that represent the output ports of the atomic model. Each *output* port is connected to a destination *dest* element containing the name of the destination atomic model and the attribute *inport* represents the destination port.

Figure 9 The schema view of flat mapping (see online version for colours)



In order to generate the map file, we have used an XSLT transformation *mapping.xsl* which uses the following algorithm (Figure 10)

Figure 10 Algorithm of mapping transformation



The description of the transformation rules of our algorithm begins with the loading of the user's generated model. A treatment of output ports of each atomic model is done. For each output port of this atomic model, a mapping of all couplings associated with this port is made in order to find all possible destinations. If the destination is an atomic model, the latter is inserted in the file *map.xml* as a destination, if the destination is a coupled model, a deep browse is done in a recursive way until reaching an atomic model.The usage of this approach reduces the number of coupling searches in the model. This flat mapping allows us to make a global browse when loading the model instead of doing it at each event.

5.2.1.2 *Generation of the simulation tree*

The simulation tree generation *SimulationTree.xml* consists in creating *Simulators* and *Coordinators* according to the hierarchy defined by the couplings of the model. Initially, an initialisation phase is launched via the file *init.xml*. The latter plays the role of *Imessage*. It allows, via its schema (Figure 11), to initialise all models. Two types of initialisation are possible:

- Model: this element is composed of the attribute *name* which represents the atomic model name and the element *state* its initial state.
- Default: this element occurs if the model is not explicitly initialised with a tag *Model*. It expresses the default state of the model.

Figure 11 XSD model initialisation (see online version for colours)

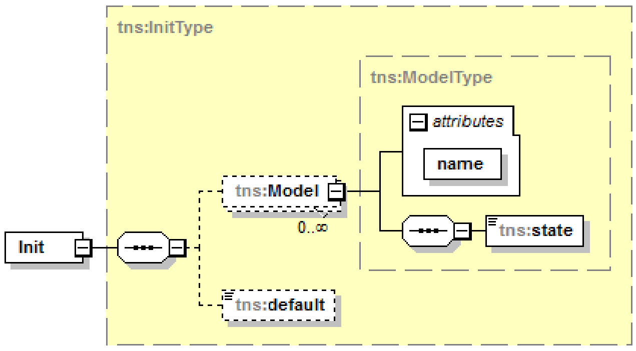
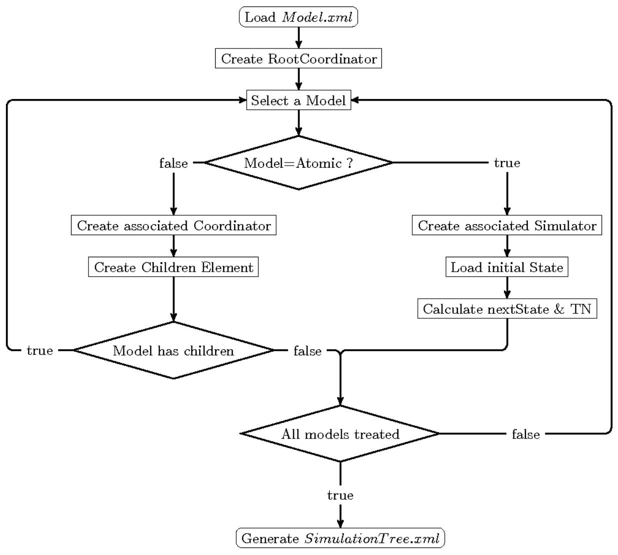


Figure 12 Algorithm of the generation of the simulation tree

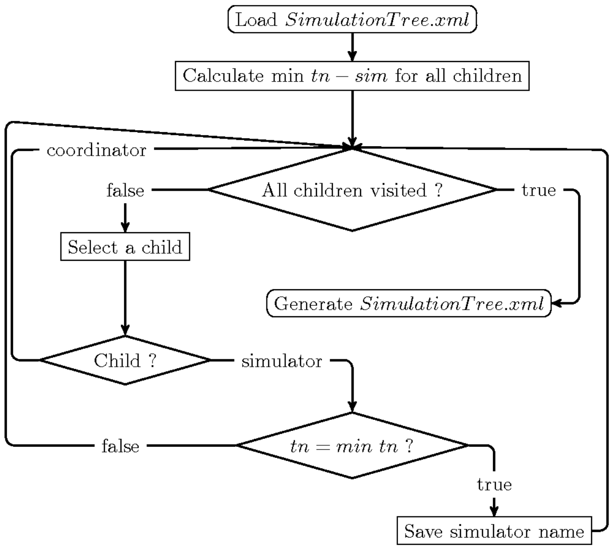


After reading the initialisation file, an XSLT transformation is performed *TreeGenerator.xsl* in order to generate the simulation tree *SimulationTree.xml*. This tree is created according to the algorithm (Figure 12) which creates an element representing a *Root* $-$ *coordinator* and associates a *Simulator* with each atomic model and a *Coordinator* with each coupled model. An initialisation message is executed for each *Simulator* with the insertion of its initial state defined in *init.xml*.

5.2.1.3 *Synchronisation of root next time event*

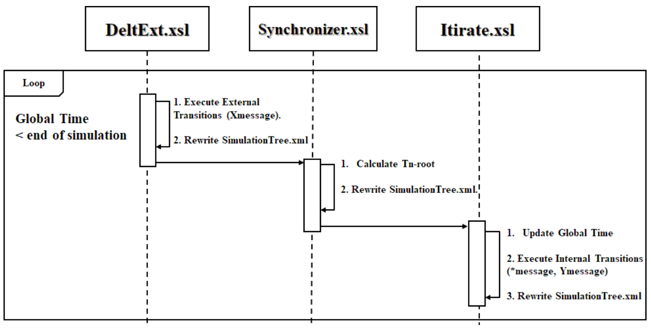
Synchronisation between *Simulators* is done by an XSLT transformation *Scheduler.xsl* described by the algorithm of Figure 13.

Figure 13 Algorithm of synchroniser transformation



The first step of the transformation gives us the minimum value of all elements *Tn-sim* of our *Simulators*. Then a search is done to extract the list of all *Simulators* concerned by this minimum value. This list of *Simulators* is inserted into the element *nextTN* of *Root-coordinator*. It is a form of sending *doneMessage*. The result of this transformation updates the simulation tree after each event processing.

Figure 14 UML Sequence diagram of simulation


DeltExt.xsl
Synchronizer.xsl
Itirate.xsl
Loop
Global Time
< end of simulation
1. Execute External Transitions (Xmessage).
2. Rewrite SimulationTree.xml
1. Calculate Tn-root
2. Rewrite SimulationTree.xml.
1. Update Global Time
2. Execute Internal Transitions (*message, Ymessage)
3. Rewrite SimulationTree.xml

5.2.2 Simulation run

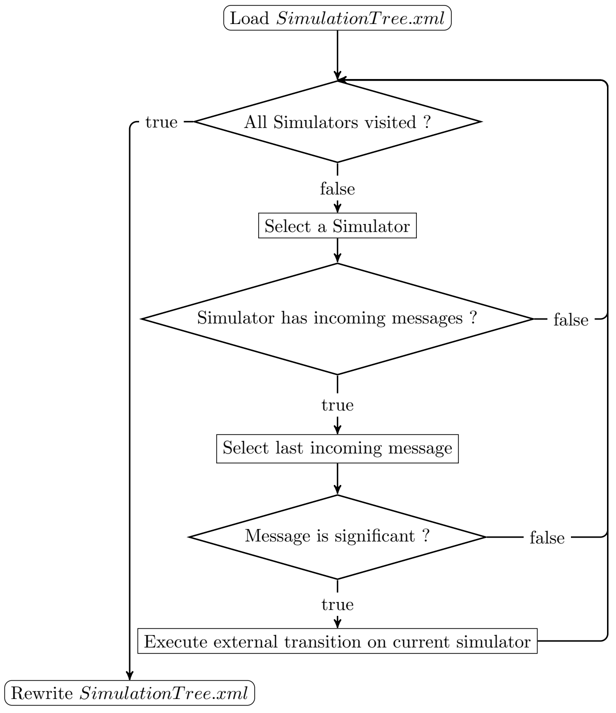
The simulation is done through a series of transformations that the simulation tree will undergo. These transformations are performed by means of three XSLT transformation files in a simulation loop (Figure 14).

5.2.2.1 External Transition transformation

For each element $Simulator$ in the simulation tree, this transformation (Figure 15) checks whether this element is a message destination in the element *output* of other $Simulators$. A browse of this tree gives a list of concerned $Simulators$. In this work, only the last element is processed. The message is processed by checking in the model whether this message causes an external transition. If true, it does the following operations:

- elment $State\text{-}sim = Atomic/deltext/ :: /NextState$
- element $Tn\text{-}sim = Atomic/ :: /timeout + RootCoordinator/GlobalTime$
- $Simulator/NextState = Atomic/deltint/ :: /NextState.$

Figure 15 Algorithm of external transition


Load $SimulationTree.xml$
All Simulators visited ?
true
false
Select a Simulator
Simulator has incoming messages ?
false
true
Select last incoming message
Message is significant ?
false
true
Execute external transition on current simulator
Rewrite $SimulationTree.xml$

5.2.2.2 Iterate

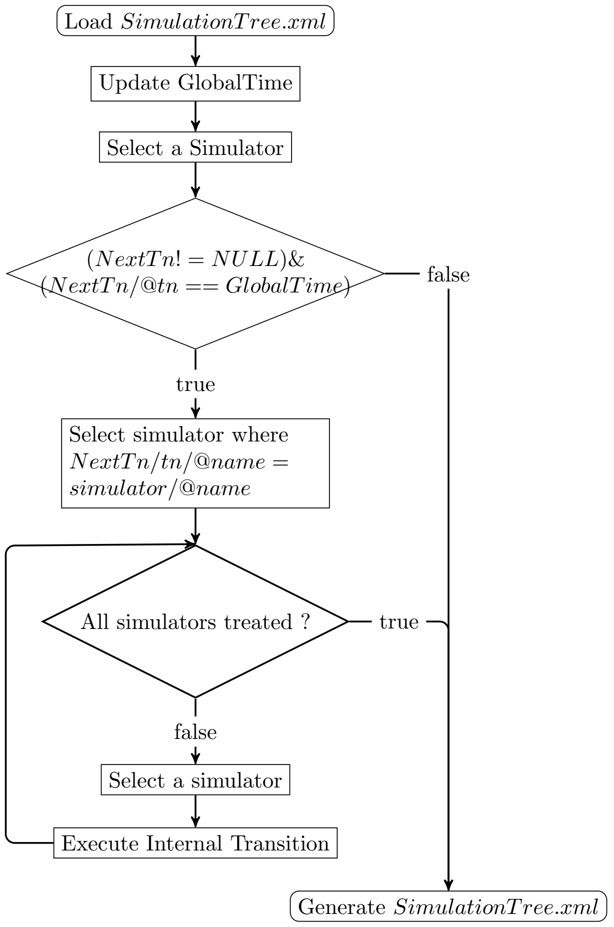
This XSLT transformation aims to ensure the global time advance of simulation. For each new $Tn\text{-}root$, an internal transition is executed by the $Simulator$ concerned (Figure 16).

This transition will result in the following operations:

- element $Simulator/State\text{-}sim = Simulator/NextState$
- element $Tn\text{-}sim = Atomic/ :: /timeout + RootCoordinator/GlobalTime$
- $Simulator/NextState = Atomic/deltint/ :: /NextState.$

If an internal transition produces an output message, the element *output* is inserted, this element represents the sending of the $Ymessage$ which will be processed by the external transition transformation.

Figure 16 Algorithm of iterate


Load $SimulationTree.xml$
Update GlobalTime
Select a Simulator
$(NextTn! = NULL)\&$ $(NextTn/@tn == GlobalTime)$
false
true
Select simulator where $NextTn/tn/@name = simulator/@name$
All simulators treated ?
true
false
Select a simulator
Execute Internal Transition
Generate $SimulationTree.xml$

6 Case study

In order to explain the functioning of our simulator, we have chosen a model as an example in order to execute it and visualise the simulation results. This model represents a traffic light positioned at crossroads. On each street, there are two opposite traffic light devices. Each of these signalling devices is supplied with a push button on which a pedestrian can click to allow crossing. In this example, we consider the following assumptions:

- When traffic light button is clicked, it switches immediately to green colour allowing pedestrian crossing.
- On the same street, both traffic lights are coupled together.
- Pushing the traffic light button by the pedestrian is a random behaviour. We suppose that a pedestrian pushes the button at $t = 19$ and $t = 39$
- The actuated button is a transient state, once clicked it triggers state change regardless of the duration of a pedestrian click.
- In order to initialise our model, we assume that the $Street1$ lights are red and $Street2$ are green.

We have four traffic light atomic models $TrafficLightNorth$, $TrafficLightSouth$, $TrafficLightEast$ and $TrafficLightWest$. All these models have this DEVS formal specification:

$$
\begin{aligned}
&TrafficLightAM = \langle X, S, Y, \delta_{int}, \delta_{ext}\lambda, ta\rangle\\
&X = \{(in, x)/in \in InPorts, x \in X_{InPorts}\}\\
&InPorts = \{inport1, inport2, inport3\},\\
&X_{InPorts} = \{Clicked, Red, Green\}\\
&Y = \{(out, y)/out \in OutPorts, y \in Y_{OutPorts}\}\\
&OutPorts = \{outport1\}, Y_{OutPorts} = \{Red, Green\}\\
&S = \{Green, Red, Yellow\}\\
&\delta_{int}(Red) = Green\\
&\delta_{int}(Green) = Yellow\\
&\delta_{int}(Yellow) = Red\\
&t_a(Green) = 10\\
&t_a(Yellow) = 5\\
&t_a(Red) = 15\\
&\delta_{ext}((Red, e), (inport1?Red)) = Green\\
&\delta_{ext}((Yellow, e), (inport1?Red)) = Green\\
&\delta_{ext}((Green, e), (inport1?Green)) = Red\\
&\delta_{ext}((Yellow, e), (inport1?Green)) = Red\\
&\delta_{ext}((Red, e), (inport2?Clicked)) = Green\\
&\delta_{ext}((Yellow, e), (inport2?Clicked)) = Green\\
&\delta_{ext}((Green, e), (inport3?Clicked)) = Red\\
&\delta_{ext}((Yellow, e), (inport3?Clicked)) = Red\\
&\lambda(Red) = (outport1!Red),\\
&\lambda(Green) = (outport1!Green)
\end{aligned}
$$

Note that the function lambda is an empty set for $TrafficLightNorth$ and $TrafficLightEast$. Then, we have for these two models:

$$
\begin{aligned}
&\lambda(Red) = \emptyset\\
&\lambda(Green) = \emptyset
\end{aligned}
$$

The pedestrian DEVS atomic model is formally specified as follows:

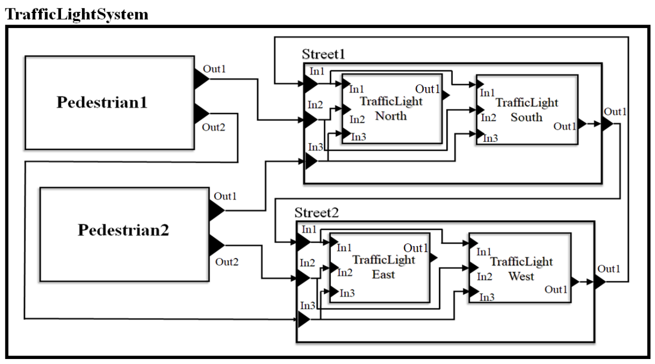
$$
\begin{aligned}
&PedestrianAM = \langle X, S, Y, \delta_{int}, \delta_{ext}\lambda, ta\rangle\\
&X = \emptyset\\
&InPorts = \emptyset\\
&Y = \{(out, y)/out \in OutPorts, y \in Y_{OutPorts}\}\\
&OutPorts = \{outport1, outport2\},\\
&Y_{OutPorts} = \{Clicked\}\\
&S = \{Clicked, Unclicked\}\\
&\delta_{int}(Unclicked) = Clicked\\
&\delta_{int}(Clicked) = Unclicked\\
&t_a(Unclicked) = \sigma_i
\end{aligned}
$$

where σ_i is random times.

$$
\begin{aligned}
&t_a(Clicked) = 0\\
&\lambda(Clicked) = (outport1!Clicked)\\
&\lambda(UnClicked) = (outport2!Clicked)
\end{aligned}
$$

Each street is considered as a coupled model, composed of two traffic light atomic models. The pedestrian atomic model is coupled to the street coupled model (Figure 17).

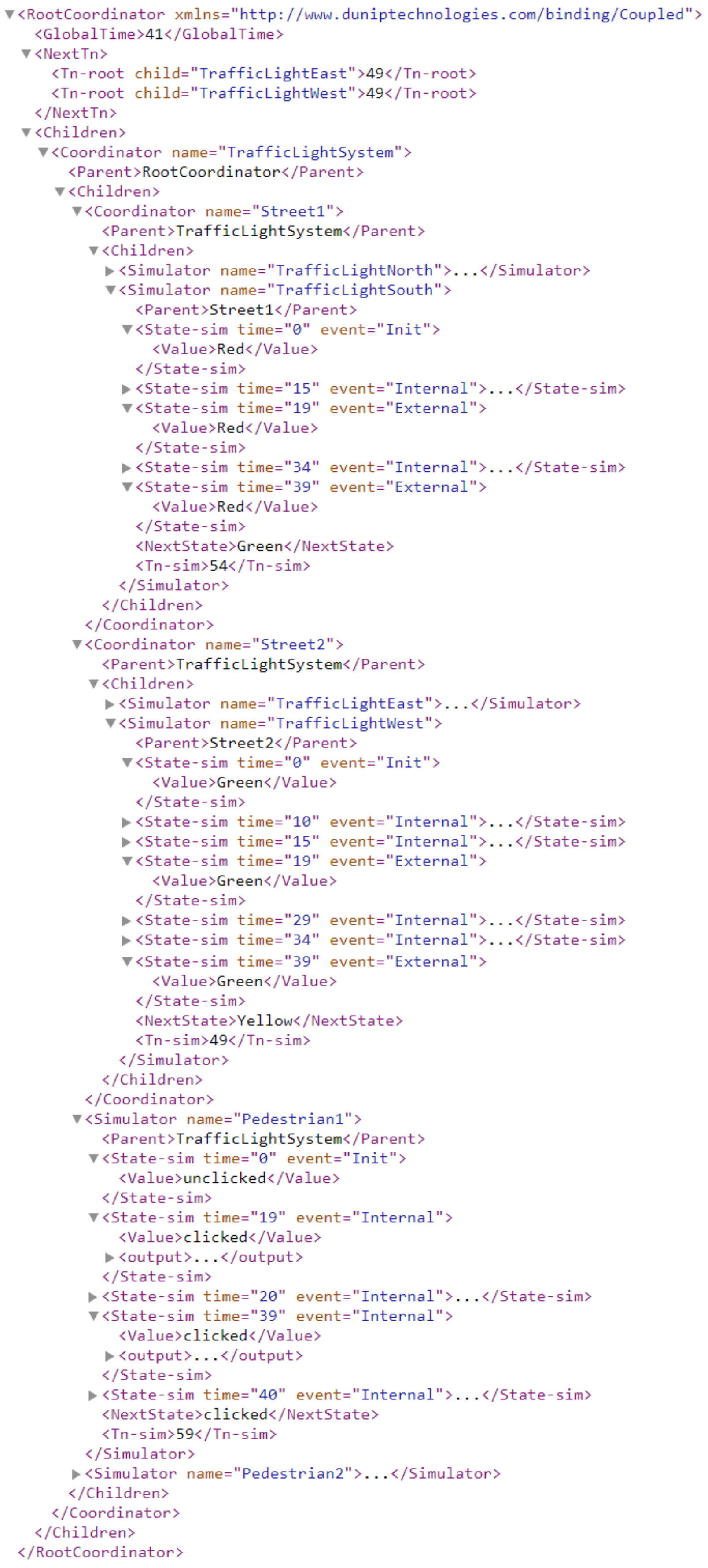
Figure 17 Traffic light DEVS coupled model



The result of the simulation of our example automatically generates an XML file $SimulationTree.xml$, in which you will find the complete structure of our model and the trace of all state changes over the simulation period. For clarity reasons, a snapshot at a fixed time in the simulation of the system is given (Figure 18).

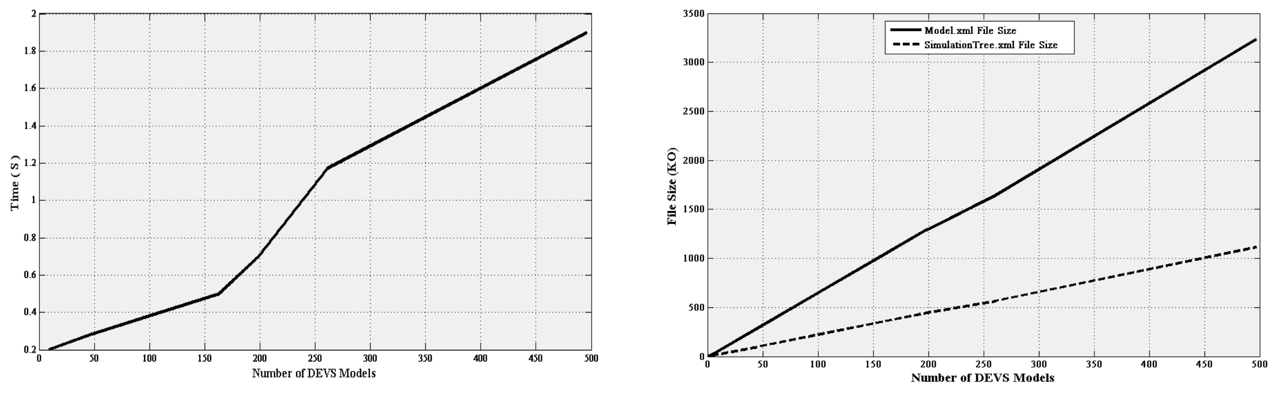
However, it should be noted that the simulation file evolves with simulation and according to the information supplied by this file, we have done an evaluation of performance to see the viability of this solution.

Figure 18 Snapshot of simulation tree at $t = 39$ (see online version for colours)



```
▼<RootCoordinator xmlns="http://www.duniptechnologies.com/binding/Coupled">
   <GlobalTime>41</GlobalTime>
 ▼<NextTn>
     <Tn-root child="TrafficLightEast">49</Tn-root>
     <Tn-root child="TrafficLightWest">49</Tn-root>
   </NextTn>
 ▼<Children>
   ▼<Coordinator name="TrafficLightSystem">
       <Parent>RootCoordinator</Parent>
     ▼<Children>
       ▼<Coordinator name="Street1">
           <Parent>TrafficLightSystem</Parent>
         ▼<Children>
           ▶<Simulator name="TrafficLightNorth">...</Simulator>
           ▼<Simulator name="TrafficLightSouth">
               <Parent>Street1</Parent>
             ▼<State-sim time="0" event="Init">
                 <Value>Red</Value>
               </State-sim>
             ▶<State-sim time="15" event="Internal">...</State-sim>
             ▼<State-sim time="19" event="External">
                 <Value>Red</Value>
               </State-sim>
             ▶<State-sim time="34" event="Internal">...</State-sim>
             ▼<State-sim time="39" event="External">
                 <Value>Red</Value>
               </State-sim>
               <NextState>Green</NextState>
               <Tn-sim>54</Tn-sim>
             </Simulator>
           </Children>
         </Coordinator>
       ▼<Coordinator name="Street2">
           <Parent>TrafficLightSystem</Parent>
         ▼<Children>
           ▶<Simulator name="TrafficLightEast">...</Simulator>
           ▼<Simulator name="TrafficLightWest">
               <Parent>Street2</Parent>
             ▼<State-sim time="0" event="Init">
                 <Value>Green</Value>
               </State-sim>
             ▶<State-sim time="10" event="Internal">...</State-sim>
             ▶<State-sim time="15" event="Internal">...</State-sim>
             ▼<State-sim time="19" event="External">
                 <Value>Green</Value>
               </State-sim>
             ▶<State-sim time="29" event="Internal">...</State-sim>
             ▶<State-sim time="34" event="Internal">...</State-sim>
             ▼<State-sim time="39" event="External">
                 <Value>Green</Value>
               </State-sim>
               <NextState>Yellow</NextState>
               <Tn-sim>49</Tn-sim>
             </Simulator>
           </Children>
         </Coordinator>
       ▼<Simulator name="Pedestrian1">
           <Parent>TrafficLightSystem</Parent>
         ▼<State-sim time="0" event="Init">
             <Value>unclicked</Value>
           </State-sim>
         ▼<State-sim time="19" event="Internal">
             <Value>clicked</Value>
           ▶<output>...</output>
           </State-sim>
         ▶<State-sim time="20" event="Internal">...</State-sim>
         ▼<State-sim time="39" event="Internal">
             <Value>clicked</Value>
           ▶<output>...</output>
           </State-sim>
         ▶<State-sim time="40" event="Internal">...</State-sim>
           <NextState>clicked</NextState>
           <Tn-sim>59</Tn-sim>
         </Simulator>
       ▶<Simulator name="Pedestrian2">...</Simulator>
       </Children>
     </Coordinator>
   </Children>
 </RootCoordinator>
```

Figure 19 Performance of XML-DEVS modelling and simulation, (a) initialisation time (b) file size evolution



(a) (b)

7 Performance evaluation

In order to evaluate the performance of our simulator, we have used the model presented above (Traffic Light System) in several combinations to simulate the behaviour of several traffic lights at the same time. The simulated time was set at 200 units of time. Here are some results (Figure 19).

The initialisation time of all models is represented by Figure 19(a). It shows that the average time to initialise a model is 0.0038 seconds. The initialisation time of all models increases with the number of simulated models.

The analysis of Figure 19(b) shows that the representation of a model (solid line) has an average size of 5.5 Ko representing the different states by which a model is passed. The trace generated by the simulator (dashed line)occupies an average size of 2.2 Ko per model. Here also, we see that the size of our simulation tree file increases according to the number of the models.

XML is adapted as part of our work because on the one hand, the transformation part is portable, i.e., XSLT transformation rules can run with a single call regardless of the programming language used, and on the other hand, XML has a data validation mechanism. The flaw in this work is the growth of the size of the 'model' and 'simulation' files generated by this implementation.

8 Conclusions and future work

Based on systems theory, the DEVS formalism provides a solid basis for modelling and simulation of discrete event systems. The formalism is sufficiently general and successfully applied to several application domains. Different teams of researchers have, depending on their needs, created several DEVS environments in order to code their models in specific programming languages. Indeed, whether these models are coded or not in the same language, each of the DEVS-oriented environments defines its own technical constraints, rules and programming conventions to translate and implement the DEVS formalism. To associate simulation algorithms and try to reuse either model or its partial part from a specific programming language into another is a hard task and requires a great effort. This lack of interoperability and reuse of DEVS models has obliged researchers to propose several approaches. This work tackles the problem of model reuse of classic DEVS formalism by proposing an XML-based modelling approach, which is independent of a particular programming language. To simulate our DEVS models, an abstract DEVS simulator is presented based on XSLT transformations. Our DEVS models are validated by XML Schema Definition, where the simulation is presented as an XML tree keeping the simulation trace over time. Thus modelling and simulation DEVS models are described in a purely textual manner. The goal of this work is twofold. The first goal is to make our schema capable of operating easily with existing or even future simulation software implementation. Then, the modeller can define a model without knowing a particular programming language of a specific modelling and simulation tool. The second goal consists of defining an alternative of simulating DEVS models in XML by generating a simulation tree. In this work we have presented the different XML schemas that represent the structure of the models as well as the various transformation components that guarantee the execution of the simulation. We have started by defining the structure of the XML file that represents the DEVS models expressed with XML schemas (XSD). Both atomic and coupled models are specified. Once the modelling part of the formalism has been realised, we proceed to the simulation of the models. This simulation is based on applying XSLT stylesheets to the models XML files. The proposed solution does not conflict with other research in the area of XML-based DEVS modelling. Instead, it can be seen as an alternative to the transformation of a model to a programming language representation, which is the predominant approach found in XML-based simulation modelling. An advantage of this solution is to provide the resulting simulation of DEVS models in a textual document which can be used for other purposes by other softwares. The most important drawback of our XML-based simulation tracking is a decreased performance because the transformation processor accepts only physical files as input instead of variables because these transformations are stateless. In fact, the simulation tree of XSLT models transformations is significantly slower than simulating the same model written in a programming language. Since classic DEVS modelling and simulation are presented, an automatic writing of the models in XML file is not supported, the modeller must write the XML file, once he finishes, he proceeds to load these files for the simulation. We aim to develop a graphical editor to assist the user in generating the model XML file. Another work aims to add conditions on ports and state variables and present them as a tree in order to inject them into our XML simulation tree. Generalise the solution to cover the other variants of DEVS formalism by developing a framework based on XSLT. The idea is to automatically translate the DEVS XML models into several DEVS platforms allowing the interoperability between different simulation software. However, a standard formalisation of DEVS will facilitate us this task.

References

Bazoun, H., Ribault, J., Zacharewicz, G., Ducq, Y. and Boyé, H. (2016) 'SLMTOOLBOX: enterprise service process modeling and simulation by coupling DEVS and services workflow', *International Journal of Simulation and Process Modelling*, Vol. 11, No. 6, pp.453–467.

Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. and Cowan, J. (2006) *Extensible Markup Language (XML) 1.1*, 2nd ed., W3C Recommendation 16 August 2006, Technical report, Copyright 2006 W3C (MIT, INRIA, Keio) [online] http://www.w3.org/TR/2006/REC-xml11-20060816/ (accessed 30 September 2019).

Casas, P.F. (2015) 'Transforming classic discrete event system specification models to specification and description language', *Simulation*, Vol. 91, No. 3, pp.249–264.

Fishwick, P.A. (2002) 'XML-based modeling and simulation: using XML for simulation modeling', *Proceedings of the 34th Conference on Winter Simulation: Exploring New Frontiers, Winter Simulation Conference*, pp.616–622.

Garredu, S. (2013) *Meta-Modeling Approach and Model Transformations in the Context of Modeling and Simulation with Discrete Events: Application to the DEVS Formalism*, PhD thesis, University Corse-Pascal Paoli, France.

Hollmann, D.A., Cristia, M. and Frydman, C. (2015) 'CML-DEVS: a specification language for DEVS conceptual models', *Simulation Modelling Practice and Theory*, Vol. 57, pp.100–117.

Hong, K. and Kim, T. (2006) 'DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems', *Information and Software Technology*, Vol. 48, pp.221–234.

Janoušek, V., Polášek, P. and Slavíček, P. (2006) 'Towards DEVS meta language', *ISC 2006 Proceedings*, pp.69–73.

Luh, C-J. (1994) 'Modelling and simulation in an object-oriented environment', *Information and Software Technolog*, Vol. 36, No. 6, pp.343–352.

Martín, J., Mittal, S., López-Peña, M. and de la Cruz, J.M. (2007) 'A W3C XML schema for DEVS scenarios', *Proceedings of the 2007 Spring Simulation Multiconference – Volume 2*, pp.279–286, Society for Computer Simulation International.

Meseth, N. (2011) *XML-based DEVS Modeling and Interpretation*, PhD thesis, University of Osnabrueck.

Mittal, S. (2007) *DEVS Unified Process for Integrated Development and Testing of Service Oreinted Architectures*, PhD thesis, The Graduate College, Univerity of Arizona.

Mittal, S. and Douglass, S.A. (2012) 'DEVSML 2.0: the language and the stack', *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, p.17, Society for Computer Simulation International.

Mittal, S. and Risco-Martín, J.L. (2017a) 'DEVSML 3.0 stack: rapid deployment of DEVS farm in distributed cloud environment using microservices and containers', *Proceedings of the Symposium on Theory of Modeling & Simulation*, p.19, Society for Computer Simulation International.

Mittal, S. and Risco-Martín, J.L. (2017b) *Netcentric System of Systems Engineering with DEVS Unified Process*, Taylor and Francis Group, CRC Press, USA.

Mittal, S., Risco-Martín, J.L. and Zeigler, B.P. (2007) 'DEVSML: automating DEVS execution over soa towards transparent simulators', *Proceedings of the 2007 Spring Simulation Multiconference – Volume 2*, pp.287–295, Society for Computer Simulation International.

Peixoto, T.A., de Assis Rangel, J.J., de Oliveira Matias, I., da Silva, F.F. and Tavares, E.R. (2017) 'Ururau: a free and open-source discrete event simulation software', *Journal of Simulation*, Vol. 11, No. 4, pp.303–321.

Santucci, J.F. and Capocchi, L. (2014) 'Fuzzy discrete-event systems modeling and simulation with fuzzy control language and DEVS formalism', *Sixth International Conference on Advances in System Simulation (SIMUL2014)*, pp.250–255, Citeseer.

Seo, C. (2009) *Interoperability between DEVS Simulators Using Service Oriented Architecture and DEVS Namespace*, PhD thesis, University of Arizona, USA.

Tendeloo, Y.V. and Vangheluwe, H. (2017) An Evaluation of DEVS Simulation Tools, *Simulation*, Vol. 93, No. 2, pp.103–121.

Wainer, G.A. and Mosterman, P.J. (2010) *Discrete-Event Modeling and Simulation: Theory and Applications*, Taylor and Francis Group, CRC Press, USA.

Wang, Y-H. and Lu, Y-C. (2002) 'An XML-based DEVS modeling tool to enhance simulation interoperability', *Proceeding 14th European Simulation Symposium*.

Zeigler, B.P. (1976) *Theory of Modeling and Simulation*, Wiley & Sons, USA.

Zeigler, B.P., Nutaro, J.J. and Seo, C. (2017) 'Combining DEVS and model-checking: concepts and tools for integrating simulation and analysis', *International Journal of Simulation and Process Modelling*, Vol. 12, No. 1, pp.2–15.

Zeigler, B.P., Praehofer, H. and Kim, T.G. (2000) *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed., Academic Press, USA.