DEVSML 2.0: The Language and the Stack

Saurabh Mittal
L-3 Communications, Air Force Research Laboratory,
Wright-Patterson AFB, OH 45433 USA
Saurabh.Mittal@L-3com.com

Scott A. Douglass
Air Force Research Laboratory,
Wright-Patterson AFB, OH 45433 USA
Scott.Douglass@wpafb.af.mil

Keywords: DSL, DEVSML, SOA, FDDEVS, Natural Language, Xtext, EMF, interoperability

Abstract

This paper presents a revised version of DEVSML stack. The earlier version introduced the concept of transparent simulators in a netcentric domain. This version of DEVSML 2.0 stack introduces the transparent modeling concept and how a platform independent DEVS Modeling Language based on Finite Deterministic DEVS can help achieve model interoperability. Further, the new stack opens up the DEVS framework [1] to customized Domain Specific Languages (DSLs) and facilitates DEVS adoption to a much wider audience. The paper describes the EBNF grammar for DEVSML language and a natural language DEVS DSL that is semantically anchored to the described language.

1. INTRODUCTION

This paper presents a platform independent DEVS Modeling language based on Finite Deterministic DEVS [2] and revises the earlier developed DEVSML framework [3,4]. The proposed DEVSML 2.0 framework has two pieces i.e. the stack and the language itself. The earlier stack realized the transparent simulation framework with DEVS/SOA [5,6] and focused more on the simulation layer. This paper is geared towards the modeling layer.

A domain specific language (DSL) is a dedicated language for a specific problem domain and is not intended to solve problems outside it. For example, HTML for web pages, Verilog and VHDL for hardware description, etc. are DSLs for very specific domains. A DSL can be textual or a graphical language or a hybrid one. A DSL builds abstractions so that the respective domain experts can specify their problem well suited to their domain understanding without paying much attention to the general purpose computational programming languages such as C, C++, Java, etc. which have their own learning curve. The notion of domain specific modeling arises from this concept and the DSL designers are tasked with creating a domain specific modeling language. If a DSL is also meant for simulation purposes, then one more task of mapping a specific DSL to a general purpose computational language is also on the cards. In this paper, we propose a DEVS DSL as a component of DEVSML 2.0. The proposed stack also integrates the transparent modeling framework with the inclusion of domain specific languages (DSLs) and various transformations. We describe how platform independent DSLs can be transformed in this framework and finally into the DEVS formalism [1]. Decoupling the model from the simulation platform has many benefits as it allows the modeler to construct models in a platform of his choice. The ability to execute DEVS models in multiple platforms has already been achieved. This ability provides a solution to scale, integration and interoperability [4-9]. Having a process to transform any DSL to DEVS components, especially to the DEVSML platform independent specification, then has obvious advantages.

The paper starts with the foundation for component-based modeling and simulation framework. In Section 2, it provides an overview of DEVS. Section 3 extends the older DEVSML stack to incorporate Domain Specific Languages (DSLs) that are platform independent and are made executable using the Model-to-Model (M2M), Model-to-DEVSML (M2DEVSML) and Model-to-DEVS (M2DEVS) transformations. Section 4 describes the EBNF grammar for the DEVSML language along with code generation and model validation features. Section 5 describes the code generation features with the Eclipse Xtext framework [10] based on Eclipse Modeling Framework (EMF). Section 6 describes a new DSL called Natural Language DEVS (NLDEVS) that is semantically anchored in DEVSML language. Finally, the conclusions are presented in the last section.

2. DEVS FORMALISM

Discrete Event System Specification (DEVS) [1] is a formalism which provides a means of specifying the components of a system in a discrete event simulation. The DEVS formalism consists of the model, the simulator and the experimental frame as shown in Figure 1. The Model component represents an abstraction of the source system using the modeling relation. The simulator component executes the model in a computational environment and interfaces with the model using the simulation relation or the DEVS simulation protocol in the present case. The Experimental Frame facilitates the study of the source system by integrating design and analysis requirements into specific frames that support analyses of various situations the source system is subjected to.

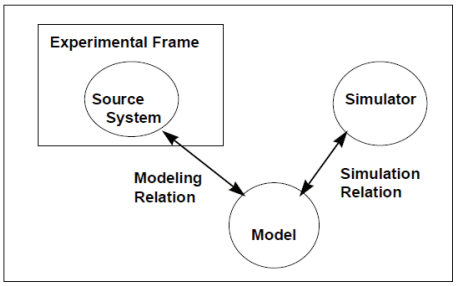

Figure 1. DEVS Framework elements

While historically models have been closely linked to the platform (such as Java, C, C++) in which the simulator was written, recent developments in platform independent modeling and transparent simulators [3,5] have allowed the development of both the models and simulators in disparate platform. Current efforts are focusing on a standardization process [11-13] wherein the simulation relation can be standardized for further interoperability.

In DEVS formalism, one must specify Basic Models and how these models are connected together. These basic models are called Atomic Models (Figure 2) and larger models which are obtained by connecting these atomic blocks are called Coupled Models (Figure 2). Each of these atomic models has inports (to receive external events), outports (to send events), a set of state variables, an internal transition function (to specify state transitions with timeouts), an external transition function (to specify state transitions on receiving external event), a confluent transition function (to specify in explicit terms whether to execute internal transition and/or external transition on the event of receiving external input when making internal transition) and a time advance function. The models specification uses or discards the message in the event to compute, deliver an output message on the outport, and make a state transition.

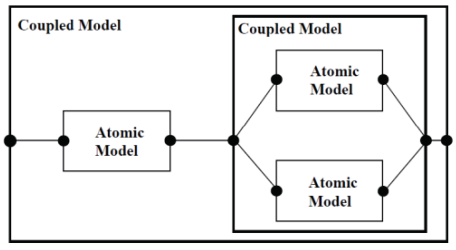

Figure 2. Atomic and Coupled models

A DEVS-coupled model designates how atomic models are coupled together and how they interact with each other to form a complex model. The coupled model can be employed as a component in a larger coupled model and can construct complex models in a hierarchical way. The specification provides components and coupling information. A Java based implementation (DEVSJAVA [14]) can be used to implement these atomic or coupled models.

3. DSL AND MODEL INTEROPERABILITY USING DEVSML 2.0 STACK

The earlier version of DEVSML stack [3,4] developed models in Java and in platform independent DEVS Modeling Language that used XML as a means for specification and transformation. The model semantics were bound together by XML and XML-based JavaML. The latest version for the stack was first proposed as a part of Air Force Research Laboratory's Large Scale Cognitive Modeling (LSCM) initiative [18]. As a part of the proposed stack DEVSML 2.0, the proposed DEVS modeling language, is based on EBNF grammar as described in the next section and is supported by DEVS middleware API. The middleware is based on DEVS M&S Standards compliant (under evaluation) API and interfaces with a net-centric DEVS simulation platform such as a service oriented architecture (SOA) that offers platform transparency. With the maturation of technologies like Eclipse Xtext 2.1 [10] and Xtend 2.0 [15], we now have extended the concept of XML-based DEVSML to a much broader scope wherein other DSLs can continue to be expressed in all their richness in a language independent manner that is devoid of any DEVS and programming language constructs (Figure 3).

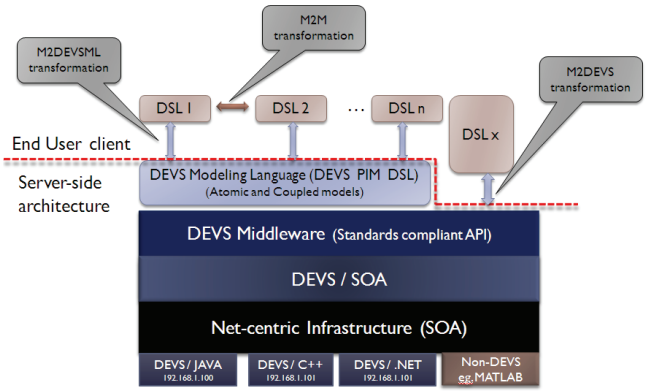

Figure 3. DEVSML 2.0 stack employing M2M and M2DEVS transformations for Model and simulator transparency

We need to make a clear distinction here that the DEVS Modeling 'language' is a DEVS modeling specification language that is anchored to DEVS simulation layer using the simulation relation in DEVS Middleware API. Consequently, a DEVSML specified model is a DEVS executable. The idea of including other DSLs at the top

layer of the stack is a major addition in the proposed DEVSML 2.0 stack. In addition, the stack in Figure 3 adds three transformations at the top layer:

1. Model-to-Model (M2M)
2. Model-to-DEVSML (M2DEVSML)
3. Model-to-DEVS (M2DEVS)

The key idea being domain specialists (the end-user) need not delve in the DEVS world to reap the benefits of DEVS framework. The end-user as indicated in Figure 3 will develop models in their own DSL and the DEVS expert along with the DSL designer will help develop the M2M and M2DEVSML transformation to give a DEVS backend to the DSL models. While M2DEVSML transformation delivers an intermediate DEVS DSL (the DEVSML DSL), the M2DEVS transformation directly anchors any DSL to platform specific DEVS. On a reverse note, a DEVS expert is ideally suited to develop DSLs in other domains as developing transformations like M2DEVS and M2DEVSML need not be negotiated with the DSL expert. A DEVS expert with DEVSML skill set can perform a dual job of both the DSL and DEVSML expert. Table 1 summarizes the state of DEVSML as viewed from the perspectives of a DEVSML expert and other DSL expert.

There are many DEVS DSLs that implement a subset of rigorous DEVS formalism. One example of DEVS DSL is XML-based Finite Deterministic DEVS (XFDDEVS) [16]. DEVSpecL [17] built on BNF grammar is another example of DEVS DSL. DSL writing tools like Xtext, Ruby, etc. focusing directly on the EBNF grammar provide a much easier foundation to develop the Abstract Syntax Tree (AST) for M2M transformations. The rich integration and code generation capabilities with open source tools like Eclipse give them strong acceptance in the software modeling community.

While M2M transformation is not really needed here, we have included it in the DEVSML 2.0 stack to close the loop per Model Driven Engineering (MDE) paradigm where meta-modeling allows such transformations. The metamodeling approach to Model Integrated Computing (MIC) brings more focus to these transformations. It also supports the Formalism Transformation Graph (FTG) with DEVS as the common denominator for multi-formalism hybrid systems modeling as mentioned by Vangheluwe in [19].

The addition of M2M, M2DEVSML and M2DEVS transformations to the DEVSML stack adds true model and simulator transparency to a net-centric M&S SOA infrastructure. The transformations yield platform independent DEVS models (PIMs) that can be developed, compared and shared in a collaborative process within the domain. Working at the level of DEVS DSL allows the models to be shared among the broad DEVS community that brings additional benefits of model integration and composability. DEVSML 2.0 stack allows DSLs to interact with DEVS middleware through an API. This capability enables the development of simulations that combine and execute DEVS and non-DEVS models [9]. This hybrid M&S capability facilitates interoperability. The scale is provided by the underlying SOA infrastructure that is largely made of virtualization technologies and utilizes platform as a service (PaaS) capabilities provided by enterprise containers such as Glassfish 4.0 [20-22].

The next sections will describe the DEVS modeling language 2.0 and how the semantics are then transformed to a DEVS executable model.

Features	**DSL Expert**	**DEVS Expert**
Domain understanding	Develops domain modeling language with domain experts	Develops M2DEVSML or M2DEVS with DSL expert
Simulation / DSL execution	Has to ensure the computational mapping and transformations taking all abstractions to code level	DEVS simulation is a 'given' as a part of DEVSML 2.0 framework
Scalability	Maybe. Has to implement an entire framework.	DEVSML compliant model is scalable at simulation layers with DEVS/SOA
Collaboration	Maybe. Has to implement an entire framework.	DEVSML compliant model is ready for collaborative development using Netcentric platform
Integration	Maybe. Has to implement an entire framework and component based infrastructure.	The DEVSML 2.0 framework with DEVS system foundation provides model integration and composition features as a part of collaborative modeling effort.
Platform neutrality	Maybe. Has to implement an entire framework.	DEVSML compliant model is platform independent both in modeling and simulation layers. The same model is executable in any DEVS simulator in any language.

Table 1: Comparing DSL Expert with DEVSML Expert

4. DEVS MODELING LANGUAGE

The DEVS Modeling Language is an extension of the earlier work on XFDDEVS which was first developed in Mittal's doctoral work [4,16]. Though XFDDEVS was a good start towards platform independent DEVS modeling, it had many shortcomings, such as, no confluent function, no multiple inputs, no multiple outputs, no complex message types and no state variables. These shortcomings are removed in the proposed language which is closer to the true DEVS formalism with some necessary abstractions. It provides a platform independent way to specify DEVS models that are transformed to platform specific language implementation in Java, C++ or any other programming language. Like any language, the DEVSML has the following keywords (Table 2):

package	import	entity
extends	coupled	models
interfaceIO	couplings	atomic
ic	eoc	eic
vars	state-time-advance	state-machine
start in	confluent	deltint
deltext	outfn	sigma
continue	reschedule	ignore-input
input-only	input-first	input-later
infinity	int	double
String	boolean	input
output	S:	S":
this	X:[]	Y:[]

Table 2. DEVSML keywords

A DEVSML file is of the extension *.fds* and the specification language contains three primary element types i.e. the Atomic, the Coupled and the Entity.

```
Type:
 Atomic | Coupled | Entity;
```

While the atomic DEVS formalism has a notion of ports (input and output), the DEVSML language has a notion of messages specified as Entity structures that are eventually transformed to port definitions. The DEVSML grammar is specified using Eclipse Xtext Extended Backus-Naur Form (EBNF) notation. The Xtext framework automatically generates a compiler, an editor, a rich validation framework, including an Eclipse *ecore* model for further extensions into the Eclipse Modeling Framework (EMF) and Eclipse Graphical Modeling Framework (GMF). For more details on Xtext EBNF capabilities, see [8]. In the following sub-sections we will look at each of the elements.

4.1. Entity

DEVS is a component-based framework where each of the components communicates using messages. When DEVS is tied to a platform specific implementation, these messages are object instances defined in the implemented language. These message objects are exchanged according to the port-value pairs specified in the atomic model structure. Consequently, the message structure is declared and defined in an atomic model, to begin with. Using the object-oriented principles, this message object structure is then reused in other atomic models. In DEVSML, as the port-value assignment is abstracted and automated, the entities are defined not as a part of the component but as a first-class citizen. These entities are then declared in atomic or coupled components for their reuse. The components exchange these entities through automated assignments as port-entity pairs. Such framework allows these entities to be used in other standardized message-exchange frameworks such as WSDL-based Service Oriented Architecture..

The EBNF specification of Entity is as follows:

```
Entity:
 'entity' name=ID
 ('extends' superType = [Entity|QualifiedName])?
 ('{' (pairs += Variable)* '}')? ;

Variable:
 Type = VarType name=ID;

VarType:
 simple = ('int'|'double'|'String'|'boolean') |
 complex = [Entity | QualifiedName] ;
```

An Entity is specified by a name, `name=ID`. It may or may not extend another entity. The expression `[Entity|QualifiedName]` means that the superType is to be specified as a `QualifiedName`, which is an Xtext construct and is of type `Entity`. It provides the full package path of the entity defined in the project. For more details on `QualifiedName`, refer Xtext manual. Each Entity contains a set of key-value pairs as `Variable` that contains a `VarType`, which may be of primitive type `('int'|'double'|'String'|'boolean')` or a complex type `[Entity | QualifiedName]`.

4.2. Atomic

While designing the Atomic DEVSML grammar we have tried to stay as close as possible to the parallel DEVS Atomic formalism. However, some abstractions were necessary. The primary abstraction as laid out in Section 4.1 is the port-value to message-entity abstraction. The other omission is the notion of elapsed time in the external event transition. The handling of elapsed time is limited to the implementation of features like 'continue' and 'reschedule' keywords, which are described later in the section. The other last piece of omission is the state transition based on

the message content. We believe that once the message content is copied over to the locally scoped variables, all the operations can be executed on the message content. However, any state change based on the message content is where the limitations of Finite Deterministic DEVS come into the picture.

The Atomic type is specified in EBNF grammar as:

```
Atomic: 'atomic' name=ID
('extends' superType=[Atomic|QualifiedName])?'{'
 'vars''{'(variables += Variable)*'}'
 'interfaceIO' '{'(msgs += Msg)*'}'
 'state-time-advance' '{'(stas += STA)*'}'
 'state-machine''{'
   'start in ' init=InitState
   (atBeh += AtomicBeh)* '}'
'}';

Msg: type = ('input'|'output')
    ref=[Entity|QualifiedName] name=ID ;
STA: name=ID timeAdv=TimeAdv;
TimeAdv:
 tav=FLOAT | inf="infinity" | var=[Variable];
InitState: state=[STA] (code=Code)?;
AtomicBeh:
 stm1=Deltext | stm2=Outfn | stm3=Deltint |
 stm4=Confluent;
Code:   '{'str=STRING '}';
```

The Atomic Type can extend from another *Atomic* type and is composed of:

- set of variables of type `Variable`. These are the locally scoped variables for the defined atomic model.
- interfaceIO specification that has a set of messages of type `Msg`. A `Msg` is either an *input* or an *output* message and is referenced as an `Entity` type. Notice that the keywords `'input'` or`'output'`are used to automate the port-entity assignments and map them to the port-value definitions in the canonical DEVS formalism.
- Set of states and the associated time-advances. Each state-time-advance pair is defined as `STA`. The time-advance `TimeAdv` can have values of either `FLOAT`, *infinity* or a `Variable` declared above in the atomic model. Using the declared variable allows the user to specify the value at execution time. This value gets assigned in the model initialization phase.
- State machine that contains the initial state `InitState` and the atomic behavior `AtomicBeh`.

Now let us look at the atomic state machine definition in more detail. The `InitState` specifies the initial state of the atomic model. The expression `state=[STA]` implies that the model references the state already defined in the construct *STA* defined earlier in the model. An element is said to have been defined when there is an ID associated with it. The next expression `(code=Code)?` implies that there may be code snippet associated with setting up of the initial state. As we shall see in a later section on code generation, the code expressed as a `STRING` is syntactically checked at run-time for any compilation errors.

The atomic DEVS formalism has *deltint*, *deltext*, *deltcon*, *lamda* functions to specify the atomic behavior and is implemented accordingly as functions in DEVSJAVA. Consequently, the DEVSML has a set of atomic behaviors and a behavior `AtomicBeh` is of the form of:

```
AtomicBeh:
 stm1=Deltext | stm2=Outfn | stm3=Deltint |
 stm4=Confluent;
```

Let's look at each of these:

```
Deltint: 'deltint' '(''S:'state=[STA] ')''=>'
'S":'target=[STA] (code=Code)?;
```

The `Deltint` specification refers a source state `'(''S:'state=[STA] ')'`and always transitions `'=>'` to the target state `'S":'target=[STA]`. Transition to the same source state is allowed.

```
Deltext: 'deltext' '(''S:'state=[STA]',''X:'
'['(in += [Msg])+ '])' '=>' 'S":'(target=[STA])?
(res = Resched | cont = 'continue')? (code=Code)?;
Resched: 'reschedule''(' setSig = SetSigma ')';
```

The `Deltext` specification refers a source state and an input message reference set `'(''S:'state = [STA]',''X:' '['(in += [Msg])+ '])'` that may transitions to a target state. The `'continue'` keyword allows the model to stay in the same source state but advance the elapsed time and redefine the sigma or time-advance as *sigma-e*. The `'reschedule'` keyword allows the resetting of time-advance of the target state. This feature overrides the time-advance of the referenced state defined earlier in `STA`. This rescheduling is characteristic of the FDDEVS specification.

```
Confluent: 'confluent' con=('ignore-input'|'input-
only'|'input-first'|'input-later') ;
```

The `Confluent` specification has four implementations as the options suggest above. During code generation, the option translates to making calls to either *deltint* or *deltext* or both in a specific order as dictated by the selection.

```
Outfn: 'outfn' '(''S:'state = [STA]')' '=>' 'Y:'
'['(out += [Msg])+ ']' (code=Code)?;
```

Finally, the `Outfn` refers a source state `'(''S:'state = [STA]')'` and maps it to an output message set `'Y:' '['(out += [Msg])+ ']'`.

All the above behavior specifications are code-assisted and validated as behavior is specified in the editor. Let's look at some of those.

4.2.1. Code-assist features for Atomic DEVSML:

Xtext is seamlessly integrated with Eclipse Modeling Framework (EMF) and the designed EBNF grammar is transformed into the native *ecore* format for Abstract Syntax Tree (AST) manipulations and code-generation. As the DEVSML editor is text-based to begin with, the code assist feature in Eclipse provides recommendations and code completion capabilities by pressing Ctrl+Space. All the references in the grammar specified as `[elementX]` are available during the model design phase. As a result, once the element is defined in early stages of the design and is identified by name=ID, it is available as a reference if the scope permits it. By default, the element is visible in the package scope, but it is highly customizable. As we shall see in the later section, we use this capability to expedite the couplings design in a Coupled model.

To reap the benefits of the code completion features, the sequential process to design an atomic DEVSML model is as follows:

1. Define entities. They may be in the same package or in a different file with a different package.
2. Declare an atomic model type with a name and import the package containing entities. Now all the entities are available as references for their reuse as message types with Ctrl+Space.
3. Start with defining atomic model variables. Here also entities are available for complex variable types.
4. Define interface inputs and outputs. Entities are available as message types
5. Define states and their time advance. All the states will be available in the state machine from this point onwards. This is important as the state-machine must not use any state for which time-advance has not been defined. The time advance can be a double, a string 'infinity' or a variable that may be assigned a value in the initialize code snippet. If there is no value specified, the default value for double is assigned which is zero.
6. Define the state machine
   a. Begin with an initial state. Select any state from the state-set defined in previous step. Ctrl+space gives you all the available states.
   b. Now you are in a position to specify either `Deltint`, `Deltext`, `Outfn` or `Confluent` behavior. Type the appropriate keyword i.e. `'deltint'`, `'deltext'`, `'outfn'` or `'confluent'` and press Ctrl+Space. Appropriate hints and options will appear that speed up the behavior specification process.

4.2.2. Run-time model validation of Atomic DEVSML

The Xtext framework provides a rich validation extension mechanism that allows writing customized validators for the defined EBNF grammar. We have currently defined the following validations that execute at run-time when the atomic behavior is being designed in the DEVSML eclipse editor:

1. Unique Model names across packages
2. No same source state for internal transitions i.e. there should be no two internal transitions with same source state.
3. Output message in Outfn must be defined in interfaceIO i.e. the message being sent in Outfn must be defined as an 'output' in atomic models' interfaceIO definition.
4. Input message in Deltext must be referenced in interfaceIO i.e. the message being received in Deltext must be defined as an 'input' in atomic models' interfaceIO definition
5. No same source state and input message set for external input transitions i.e. there should be no two external input transitions with same source state **and** identical input message set.
6. No same source state for output functions i.e. there should be only one output function associated with a specific state.

These validations provide the next level of model checking. The first level is provided by the scoping the references available as explained earlier. The transformed code after this validation is *DEVS-correct* by construction.

4.3. Coupled

The DEVSML specification of a coupled DEVS is as follows:

```
Coupled: 'coupled' name=ID
 ('extends' superType=[Coupled|QualifiedName])?'{'
 'models' '{'(components += Component)*'}'
 'interfaceIO' '{'(msgs += Msg)*'}'
 'couplings' '{' (couplings += Coupling)*'}'
 '}';
Component: AtomicComp | CoupledComp;
AtomicComp:
 'atomic' at = [Atomic | QualifiedName] name=ID;
CoupledComp:
 'coupled' cp = [Coupled | QualifiedName] name=ID;
Coupling: ic=IC | eoc=EOC | eic=EIC;
EIC: 'eic'
 'this'':' msgtype = [Msg] '->'
 dest = [Component]':' destMsgType = [Msg];
IC:'ic'
 src= [Component | QualifiedName] ':'
 msgtype =  [Msg] '->'
 dest= [Component | QualifiedName] ':'
 destMsgType = [Msg];
EOC: 'eoc'
 src = [Component] ':' msgtype = [Msg] '->'
 'this'':' destMsgType = [Msg] ;
```

The `Coupled` DEVSML type can extend from another DEVSML `Coupled` component. It is composed of:

- Set of models, of type `Component`. The `Component` can be of either type `AtomicComp` or `CoupledComp`. Note

that each has a *name*. This allows multiple components of the same type within a coupled model. For example, multiple Processor components with different name, all of type Processor.

- interface specification: Please see atomic above for detailed description
- Set of Couplings of type `Coupling`. In the original DEVS formalism, the couplings are specified by connecting port specifications from one component to other. In DEVSML, as we abstracted away from port specifications, we use the message `Msg` entity that flows between the components. This abstraction is transformed to port specification in the code-generation phase. Each `Coupling` specification is of any of the following types:
  - External Input Coupling `EIC`: This type of coupling is defined for connections originating from input interface of the coupled model to its subcomponents. Consequently, the source is identified as `'this'` keyword and the coupling allows routing of input message defined in its `'interfaceIO'` to the destination sub-model specified through `Component`.
  - Internal Coupling `IC`: This type of coupling is specified *between* the sub components as enumerated in `Component`.
  - External Ouput Coupling `EOC`: This type of coupling is specified from the contained `Component` to the outside interface of the coupled model. Consequently, the destination is specified as `'this'`.

4.3.1. Code-assist features of Coupled DEVSML

Having designed the atomic DEVSML models, the eclipse Xtext editor is further used to design the coupled DEVSML models. The coupled model may be in the same package as an atomic model or in a different package or in a different file altogether. Usage of *import* statements provides dependencies from other components. To define a coupled model following steps are executed. The code assist features are made available incrementally.

1. Specify the package and imports
2. Specify the coupled model name and any supertype of type `Coupled`
3. Specify components. The components are referenced on `'atomic'`or `'coupled'` keyword
4. Specify the `'interfaceIO'` using `'input'`or `'output'` keyword. Based on the package scope or imports the message entities are made available.
5. Specify couplings. The coupling statements begin with `'eic'`, `'ic'` or `'eoc'` keywords.
   a. *IC coupling*: Pressing Ctrl+Space will show up the available subcomponents defined in the Component section above. Pressing Ctrl+space for the Message outgoing source will show up the referenced message name defined in interfaceIO of the selected subcomponent. If there is no output interface message, then no message is available as an option. Consequently, this subcomponent cannot have any output couplings. Here the scope of the output source message has been customized. Pressing Ctrl+Space again, select the destination subcomponent model. Again, using scopes, only those components will show up in content assist that can receive the same Entity. Moving along, all the input messages of the selected destination subcomponent are made available using Ctrl+space that are of the same Entity type.
   b. *EIC coupling*: All the operations remaining same as above, the only change is that the source is always available as '`'this'` and the outgoing source message is scoped from the interfaceIO defined for the current coupled model.
   c. *EOC coupling*: All the operations remaining same as above, the change is visible in the destination component as 'this' and input destination message is scoped from the interfaceIO defined for the current coupled model.

4.3.2. Run-time model validation of Coupled DEVSML

We have currently defined the following validations for a coupled DEVSML model

1. Unique model names across packages
2. Model types for super type and component set are already filtered based on the keyword `'atomic'`or `'coupled'`
3. IC coupling:
   - Source output message must be defined in the interfaceIO of the source component as `'output'`
   - Destination input message must be defined in destination interfaceIO as `'input'` and of the same type `Entity` as source message type.
4. EIC coupling
   - Source output message must be defined in current coupled models' interfaceIO as `'input'`
   - Destination input message must be defined in destination interfaceIO as `'input'` and of the same type `Entity` as source message type.
5. EOC coupling
   - Source output message must be defined in component's interfaceIO as `'output'`
   - Destination input message must be defined in the current coupled model's interfaceIO as `'output'`

When the coupled models are constructed using code-assist features and validated by the rules above, the coupled DEVSML models are *DEVS-correct*. The next section will describe the code generation mechasnisms within the Xtext Eclipse framework

5. DYNAMIC CODE GENERATION

Once the models are created using Xtext Eclipse based editor, the next step is to get an executable DEVS code. The DEVSML is semantically anchored in DEVS formalism and the abstractions in DEVSML are unpacked during the code generation phase. The most prominent abstraction is the port-value to message-entity mapping.

After the specification of Fds grammar, the Xtext framework generates a bunch of artifacts to specify the scope providers, validators and code generators. These are provided as functions that can be overridden to support the grammar under design. The platform specific code that takes any *.fds* file and generates DEVSJAVA code is specified in the *FdsGenerator.xtend*. This specific file is a holder for class called *FdsGenerator* that implements the *org.eclipse.xtext.generator.IGenerator* interface. It provides only one method *doGenerate(Resource resource, IFileSystemAccess fss){}* that needs to be overridden to provide our DEVSJAVA code. The entire *Fds* Abstract Syntax Tree is available to us for manipulation and code generation in the *.xtend* file.

The first step is extracting elements of specific types (Figure 4). Recall from previous section that there are 3 types in Fds grammar i.e. Entity, Atomic and Coupled. Providing complete code generator for each of them is outside the scope of this paper and will be reported in our extended article. For overview, we shall take the example of Coupled types. The procedure of extracting any element from AST (Coupled in this case) is provided in a library function:

```
for(e: resource.allContentsIterable.filter(typeof(Coupled))){
    coupledPackageName = e.eContainer.fullyQualifiedName.toString
    fsa.generateFile(
        e.fullyQualifiedName.toString.replace(".","/")+".java",
        e.compile
    )
}
```

```
def compile (Coupled e)'''
    «IF e.eContainer !=null »
        package «e.eContainer.fullyQualifiedName»;
    «ENDIF»
    import java.util.*;
    import GenCol.*;
    import simView.*;
    import genDevs.modeling.*;

    public class «e.name» extends«IF e.superType != null» «e.superType.fullyQualifiedName»«ELSE» ViewableDigraph«ENDIF»{

        public «e.name»(){
            this("«e.fullyQualifiedName»");
        }

        public «e.name»(String name){
            super(name);
            «FOR f: e.components»
            «f.compile»
            «ENDFOR»

            «FOR f: e.msgs»
            «IF f.type.equals("input")»addInport("in«f.name.toFirstUpper»");«ENDIF»
            «IF f.type.equals("output")»addOutport("out«f.name.toFirstUpper»");«ENDIF»
            «ENDFOR»

            «FOR f: e.msgs»
            «IF f.type.equals('input')»addTestInput("in«f.name.toFirstUpper»", new «f.ref.fullyQualifiedName.toString»());«ENDIF»
            «IF f.type.equals('input')»addTestInput("in«f.name.toFirstUpper»", new «f.ref.fullyQualifiedName.toString»(),1);«ENDIF»
            «ENDFOR»

            «FOR f: e.couplings»
            «f.compile»
            «ENDFOR»
        }

    }
'''
```

Figure 5. Coupled Code generation in FdsGenerator.xtend

Then using the xtend language syntax, we can define a function that compiles the extracted entity. Figure 5 shows the template for a Coupled DEVSJAVA file. The grey areas show actual tab spaces that the generated *.java* file will contain. Consequently, other elements such as Entity and Atomics are generated and package declarations are maintained. As the Xtext framework is seamlessly integrated with the Eclipse Java framework, any *save* process (by invoking the Ctrl+S) in the editor invokes the *doGenerate()* function that generates the entire code-base. As a result, one can dynamically view the generated code from the abstract DEVSML *.fds* specification. As we will see in the next section, this capability is one of the very important features in our current selection of Xtext framework, the logic code specified in the CODE element in Atomic files is readily checked for syntactical errors. Since the generated code is already in an Eclipse java project, the

modeler is informed of any compilation errors that DEVSML model might introduce.

The generated codebase also contains a *Main.java* file that invokes the Simview DEVSJAVA viewer as the generated java code-base is available compiled.

6. EXAMPLE

To illustrate the capability of the DEVSML, we use the classic EFP hierarchical model. EFP contains two models viz. coupled *EF* and atomic *Processor*. The coupled EF internally contains two models viz. atomic *Genr* and atomic *Transd*. The Genr sends entity Job at a periodic rate. The processor Proc receives the generated *Job* and gets busy processing it. On completion it reports it to the transducer *Transd* that keeps a count of generated as well as processed job. The *Transd* and *Genr* are part of the Experimental Frame *EF*. The Transd has an observation time after which it reports the throughput.

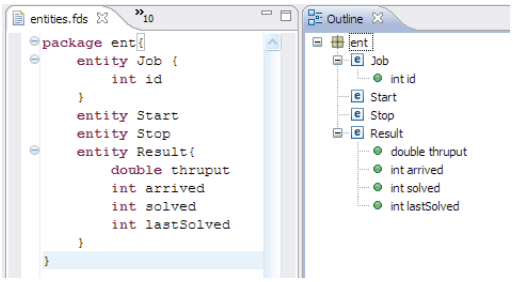

Figure 5. DEVSML Entity

The entities exchanged between the components are shown in Figure 5. The entities are in package *ent*. The outline view shows the package containment.

Figure 6 shows the Genr DEVSML model in Eclipse workbench. The first column shows the *TestFds2* Java project. The second column shows the *gpt.fds* file and the package *gpAtomics* containing the atomic *Genr* DEVSML specification. Notice the keyword highlighting based on Fds EBNF grammar and state-machine specification statements starting with keywords **deltint**, **deltext**, **outfn** etc. The third column shows the autogenerated *Genr.java* file that is synchronized with the *Genr.fds* file. Every Save operation results in code sync from *.fds→.java*. Notice the generated java code in the third column. It synthesizes the inports, the outports and the test-inputs. Further, the state-time-advances are stored in a hashtable and the time advance for a state is retrieved from it at runtime. The last column on the right shows the hierarchical Outline structure of the *Genr.fds* file with rich icons and other supplemental information. Similarly, the processor Proc and the transducer *Transd* are designed using the workbench. The generated files are held in *src-gen* folder that is compiled using the *DEVSJAVA31.jar* on the project classspath.

Figure 7 shows the two coupled models EF and EFP. The figures are self-explanatory. The autogenerated code can be seen in the third column along with the Outline in the last column. Figure 8 shows the fully functional model along with tooltip for *Transd*.

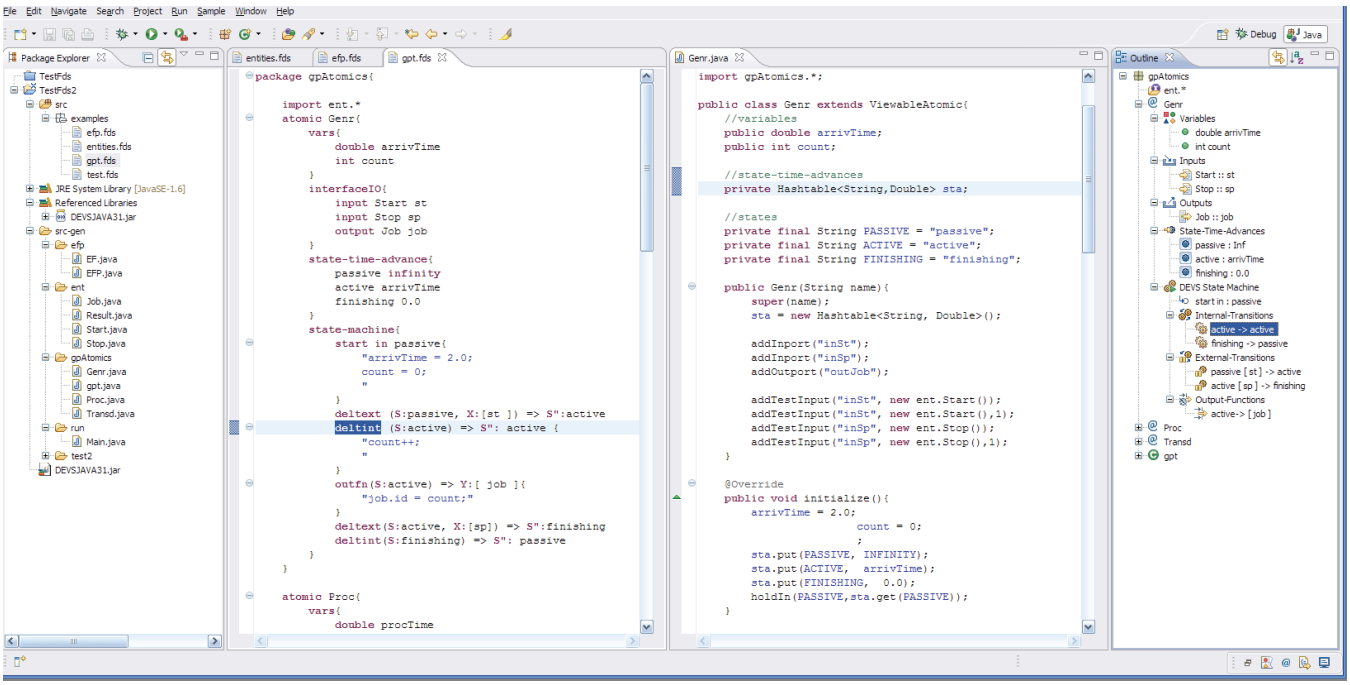

Figure 6. Atomic DEVSML Genr and the generated artifacts

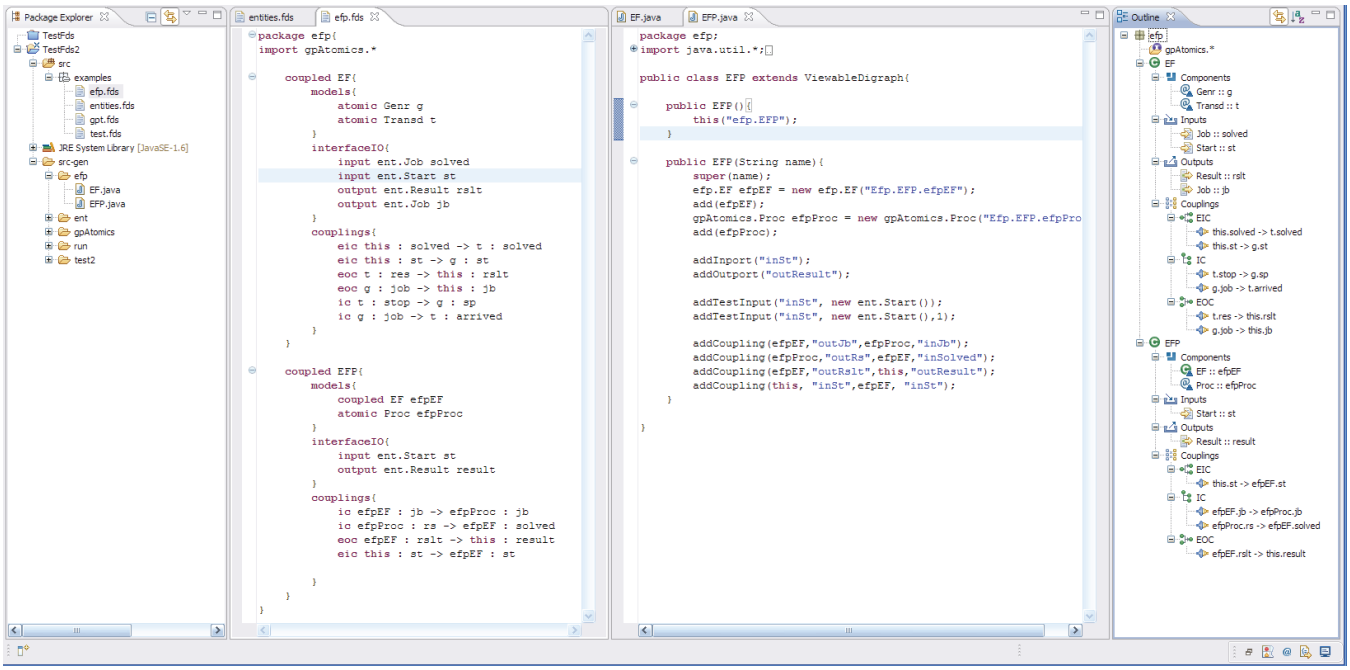

Figure 7. Coupled DEVSML EFP and the generated artifacts

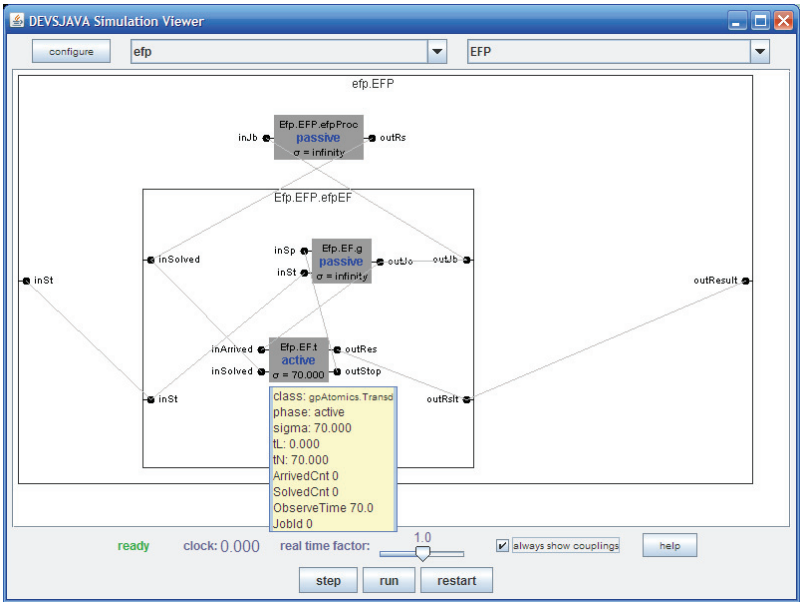

Figure 8. Fully functional autogenerated DEVSJAVA code in Simview

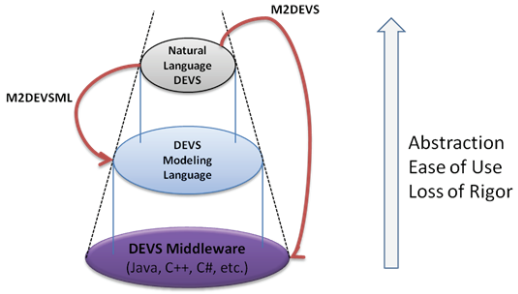

Figure 9. Moving up the abstraction and loss of rigor

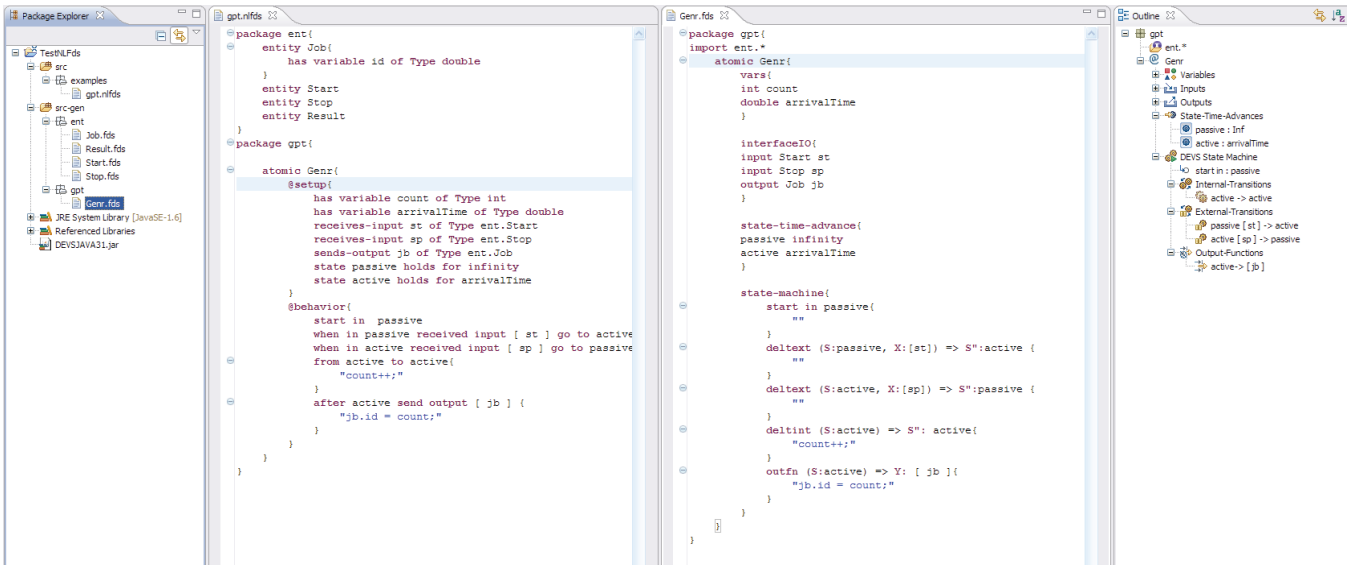

Figure 10. NLDEVS description of atomic Genr

7. DSL FOR NATURAL LANGUAGE DEVS

Now, having described the DEVSML platform independent language that is semantically anchored to DEVSJAVA, we would like to go a step further and design another DSL that can be transformed to DEVSML. One such example has been demonstrated in our earlier work [18] in which a cognitive domain DSL was transformed using M2DEVSML and DEVS homomorphism concepts [1]. Such transformation (M2DEVSML) is according to conceptual framework laid out in Section 3. Providing details about the M2DEVSML transformation is outside the scope of this article and will be reported in our extended article.

We designed another grammar based on structured natural language and call it Natural Language DEVS (NLDEVS). The earlier version of NLDEVS was done in [16] which was bounded by the XFDDEVS specification. Consequently, it had a lot of inherent limitations. NLDEVS is mentioned here to establish the concrete nature of the DEVSML 2.0 stack. NLDEVS can be directly mapped into either the DEVSML using M2DEVSML transformation or into the DEVS middleware using M2DEVS transformation. The relationship between NLDEVS, DEVSML and DEVS middleware is summarized in Figure 9. Figure 10 shows result of M2DEVSML transformation where NLDEVS is transformed to DEVSML. A sample of atomic NLDEVS is shown in Figure 10 that shows the state-machine description of the *Genr* atomic component.

As can be easily seen, the keywords of NLDEVS are totally different from that of DEVSML. The NLDEVS editor is replete with code completion and model checking such that the designed model is *DEVS-correct*. The autogenerated DEVSVML *Genr.fds* can be seen in the right column and any validation checks at the DEVSML layer are made visible during the incremental *Save* operations.

8. CONCLUSIONS

The work on platform independent DEVS specification began with XFDDEVS with Mittal's doctoral work. The XML-based DEVS was a significant development towards the development of Netcentric DEVS Unified Process. However, XFDDEVS had many shortcomings, such as, no confluent function, no multiple inputs, no multiple outputs, no complex message types and no state variables. The earlier proposed DEVSML stack also used XML-based transformations and used JavaML to specify the XML-based DEVS logic. This again was not true platform independence. With the proposed DEVSML 2.0 stack and the language, we have covered all the shortcomings of our earlier work and with advanced code generation framework like Xtext, we have shown how a DEVS and non-DEVS DSL can be effectively used to write a fully functional DEVS executable code. We have also shown in our earlier work the use of homomorphisms in transformation of non-DEVS DSLs to a DEVS DSL and also shown that a DEVS simulation is platform transparent with DEVS/SOA. With the capability presented in this paper, we are now in a position to achieve model transparency with M2DEVS and M2DEVSML transformations. It should be noted that the complexity of these transformations are on a case by case basis depending on the abstractions of a particular DSL. However, as a DEVS expert, the design of such

transformations will reap far reaching benefits to the execution, scalability, interoperability, integration and collaborative capabilities for any DSL. Further with DEVS as common denominator in multi-formalism hybrid modeling effort, DEVSML 2.0 framework provides many advantages. We highlighted the roles of a DSL expert and a DEVS expert in the context of a DSL and a simulation DSL. We also showed how a natural language based DSL can be semantically anchored to the designed DEVS modeling language. This paper has made two major contributions. First, the DEVS modeling language called DEVSML that is very close to the true DEVS formalism and second, the DEVSML2.0 stack that integrates DSLs to DEVS transparent simulation infrastructure using M2DEVS and M2DEVSML transformations.

Acknowledgements

This work has been supported by AFOSR grant # 10RH05COR

References

[1] Zeigler, BP, Kim, TG and Praehofer, H, "Theory of Modeling and Simulation" New York, NY, Academic Press, 2000

[2] Hwang, MH, Zeigler, BP, "Reachability Graph of Finite Deterministic DEVS", IEEE Transactions on Automation Science and Engineering, 2007

[3] Mittal, S, Martin, JLR, Zeigler, BP, "DEVSML: Automating DEVS Simulation over SOA using Transparent Simulators", DEVS Symposium, 2007

[4] Mittal, S, "DEVS Unified Process for Integrated Development and Testing of Service Oriented Architectures", Ph. D. Dissertation, University of Arizona, 2007

[5] Mittal, S, Martin, JLR, Zeigler, BP, "DEVS-Based Web Services for Net-centric T&E", Summer Computer Simulation Conference, 2007

[6] Mittal, S, Martin, JLR, Zeigler, BP, "DEVS/SOA: A Cross-Platform Framework for Net-Centric Modeling and Simulation in DEVS Unified Process", SIMULATION: Transactions of SCS, Vol. 85, No. 7, pp. 19-450, 2009

[7] Zeigler, BP, Mittal, S & Hu, X, "Towards a formal standard for interoperability in M&S/system of systems integration" *Proc. GMU-AFCEA Symposium on Critical Issues in C4I,* 2008

[8] Mittal, S, Zeigler, BP, Martin, JLR, "Implementation of Formal Standard for Interoperability in M&S/System of Systems Integration with DEVS/SOA", International Command and Control C2 Journal, Special Issue: Modeling and Simulation in Support of Network-Centric Approaches and Capabilities, Vol. 3, No. 1, 2009

[9] Martín, JLR, Moreno, A, Aranda, J, Cruz, JM, "Interoperability between DEVS and non-DEVS models using DEVS/SOA". *In SpringSim'09: Proceedings of the 2009 spring simulation multiconference*: 1-9 (San Diego, CA, USA, 2009)

[10] Xtext Language Development Framework accesible at: http://www.eclipse.org/Xtext/

[11] Wainer, G, Al-Zoubi, K, Dalle, O, Hill, D, Mittal, S, Martin, JLR, Sarjoughian, H, Touraille, L, Traore, M, Zeigler, BP, "DEVS Standardization: Ideas, Trends and Future", chapter in "Discrete Event Modeling and Simulation: Theory and Applications", 2010, CRC Press.

[12] Wainer, G, Al-Zoubi, K, Dalle, O, Hill, D, Mittal, S, Martin, JLR, Sarjoughian, H, Touraille, L, Traore, M, Zeigler, BP, "Standardizing DEVS Model Representation", chapter in "Discrete Event Modeling and Simulation: Theory and Applications", 2010, CRC Press.

[13] Wainer, G, Al-Zoubi, K, Dalle, O, Hill, D, Mittal, S, Martin, JLR, Sarjoughian, H, Touraille, L, Traore, M, Zeigler, BP, "Standardizing DEVS Simulation Middleware", chapter in "Discrete Event Modeling and Simulation: Theory and Applications", 2010, CRC Press

[14] DEVSJAVA: http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip

[15] Xtend, accessible at http://www.eclipse.org/Xtext/#xtend2

[16] Mittal, S, Zeigler, BP, Hwang, MH, "XFDDEVS: XML-Based Finite Deterministic DEVS", last accessed Jan 2011 at: *http://www.duniptechnologies.com/research/xfddevs/*

[17] Hong. KJ, Kim, TG, "DEVSpecL-DEVS specification language for modeling, simulation and analysis of discrete event systems," *Information and Software Technology*, Vol. 48, No. 4, pp. 221 - 234, Apr., 2006

[18] Mittal S, Douglass, SA., "From Domain Specific Languages to DEVS Components: Applications to Cognitive M&S", Spring Simulation Multiconference, April 2011, Boston.

[19] Vangheluwe, HLM, "DEVS as a common denominator for multi-formalism hybrid systems modeling", IEEE International Symposium on Computer Aided Control System Design, 2000.

[20] Wiki: Platform as a Service, http://en.wikipedia.org/wiki/Platform_as_a_service, last accessed Feb 2012

[21] Mittal, S, Martin, JLR, "Netcentric System of Systems Engineering with DEVS Unified Process", CRC Press, Nov 2012

[22] Oracle Wiki: Glassfish PaaS Functional Specification, https://wikis.oracle.com/display/GlassFish/GlassFish+PaaS+FSD , last accessed Feb 2012

Biography

Saurabh Mittal is a research scientist at AFRL for L-3 Communications. He received both his PhD (2007) and MS (2003) in Electrical and Computer Engineering from the University of Arizona. His current research interests include executable architectures using SOA, DEVS Unified Process, executable architectures, Cognitive systems and multiplatform modeling. He can be reached at saurabh.mittal@l-3com.com.

Scott Douglass is a research psychologist with the Cognitive Models and Agents Branch of AFRL's Human Effectiveness Directorate. He received his PhD (2007) in Cognitive Psychology from Carnegie Mellon University. His current research interests include large-scale cognitive modeling, generative cognitive architectures and the modeling of situated action. He can be reached at scott.douglass@wpafb.af.mil.