From Domain Specific Languages to DEVS Components: Application to Cognitive M&S

Saurabh Mittal
L-3 Communications, Air Force Research Laboratory,
Mesa, AZ 85212 USA
Saurabh.Mittal@L-3com.com

Scott A. Douglass
Air Force Research Laboratory,
Mesa, AZ 85212, USA
Scott.Douglass@mesa.afmc.af.mil

Keywords MIC, DSL, ACT-R, M2M, DEVSML, SOA

Abstract

Air Force Research Lab (AFRL) research efforts to transition cognitive modeling from the laboratory to operational environments are finding that available architectures and tools are difficult to extend, lack support for interoperability standards, and struggle to scale. This paper describes a component-based modeling and simulation framework that exploits the Discrete Event System Specification (DEVS) formalism to eliminate these impediments. It extends the earlier DEVS Modeling Language (DEVSML) architecture and integrates Domain specific languages (DSLs). The paper discusses the framework and the transformation processes with a DSL tailored to the needs of cognitive modeling.

1. INTRODUCTION

AFRL research efforts employing cognitive modeling are growing in scale and complexity. Researchers contributing to these efforts are struggling to meet the challenges of increasing the scale of their models and integrating them into software-intensive distributed training environments. The struggle has two sources: (1) the need to specify detailed knowledge and process descriptions in our modeling frameworks; and (2) a dependence on specialized simulators in our modeling frameworks that isolates our models from standards, methods, and tools utilized by the larger systems engineering community.

An AFRL large-scale cognitive modeling (LSCM[1]) research initiative is developing solutions to these scale and interoperability challenges based on high-level languages, more specifically domain specific languages (DSLs), for describing cognitive models and simulation frameworks supporting them based on the Discrete Event System Specification (DEVS) formalism [1]. This paper discusses a modeling and simulation framework based on earlier developed DEVS Modeling Language stack [2]. We illustrate how platform independent DSLs can be transformed in this framework into the DEVS formalism. We describe a DSL specifically designed to support cognitive modeling and the procedures through which models specified in the DSL can be transformed into DEVS components that can be executed in advanced modeling and simulation platforms transparently. 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 [2, 3]. Having a process to transform any DSL to DEVS components 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 and how Model Integrated Computing (MIC) can be facilitated by DEVS. In this section we discuses a DEVS implemented on Service Oriented Architecture (SOA) execution platform. Section 3 extends the existing DEVS Modeling Language (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 a new DSL for cognitive modeling. Section 5 describes a process that transforms this new DSL to DEVS using a M2DEVSML transformation that transforms eventually to DEVS platform specific execution. Finally, the conclusions and future work is addressed.

2. FOUNDATIONS OF COMPONENT BASED MODELING AND SIMULATION FRAMEWORK

2.1. Model Integrated Computing (MIC)

The LSCM initiative is researching solutions to the scale and interoperability challenges based on Model Integrated Computing (MIC), a general modeling and systems integration paradigm [4]. MIC facilitates LSCM because it:

(1) allows cognitive modelers to specify models in DSLs tailored to the needs of cognitive modeling
(2) supports the composition of these DSLs [5]
(3) automates the integration of models specified in these DSLs into task environments or larger systems [6]

[1] **This research is being funded through AFOSR grant # 10RH05COR**

(4) provides automated model-to-model (M2M) transformation capabilities that produce executable code artifacts from models specified in these DSLs.

As our modeling ambitions grow, the inability to share significant models, to make them components of larger system of integrated and extended models, amplifies the costs of any modeling endeavor. To share and integrate our models, we must find a way to generally cast them as components in larger M&S frameworks. We are developing a cognitive M&S framework in the LSCM initiative that will allow modelers to define and execute componentized models.

From an architectural perspective, the framework consists of net-centric M&S infrastructure based on the DEVS formalism [1]. The architecture technically realizes a DSL for cognitive modeling into the proposed DEVSML stack using various model transformations. From a user perspective, the framework consists of a set of DSLs that are automatically transformed into the DEVSML and executed in a transparent M&S infrastructure.

2.2. 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.

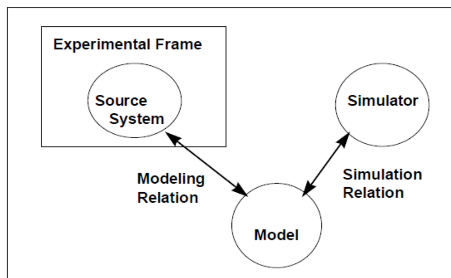

Figure 1. DEVS Framework elements

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 [2, 3] have allowed the development of both the models and simulators in disparate platform. Current efforts are focusing on a standardization process [7-9] 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.

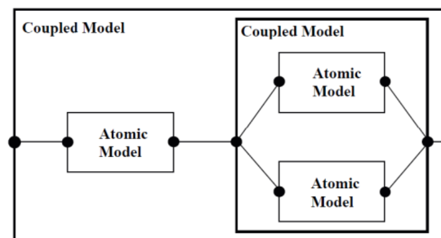

Figure 2. Atomic and Coupled models

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 [10]) can be used to implement these atomic or coupled models.

2.3. DEVS/SOA

The DEVS/SOA framework [11] is analogous to other DEVS distributed simulation frameworks like DEVS/HLA, DEVS/RMI and DEVS/CORBA [12-16] and uses web-services as the underlying technology to implement the DEVS simulation protocol. The distinguishing mark of DEVS/SOA is that it uses SOA as the network communication platform and XML as the middleware and thus acts as a basis of interoperability using XML [17]. The complete setup requires one or more servers that are capable of running DEVS Simulation Service, as shown in Figure 3.

The capability to run the simulation service is provided by the server side design of DEVS Simulation protocol supported by the latest DEVSJAVA Version 3.1 [10].

Once a DEVS model package is developed, the next step is executing the simulation as illustrated in Figure 3. The DEVS/SOA client (Figure 3) takes the DEVS models package and through the dedicated servers hosting DEVS simulation services, it performs the following operations:

1. Using the client application locate DEVS simulation servers
2. Select the Simulation resources
3. Compose your root coupled model
4. Perform Simulation on SOA
    a. Upload the models to specific IP locations i.e. partitioning
    b. Run-time compile at respective sites
    c. Simulate the coupled-model
5. Receive the simulation output at clients end

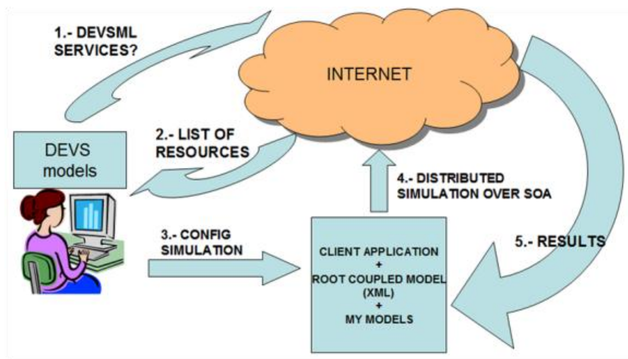

Figure 3. DEVS/SOA Execution flow

The DEVS/SOA has been used to integrate both DEVS and non-DEVS models in a single transparent simulation exercise [18].

3. DSL AND MODEL INTEROPERABILITY USING DEVSML 2.0 STACK

The earlier version of DEVSML stack [2,3] developed models in Java and in platform independent DEVS Modeling language that used XML as a means for transformation. The model semantics were bound together by XML. The latest version of the DEVSML, the language, is based on EBNF grammar 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 Xtext [19] and Xpand [20] we have now extended the concept of XML-based DEVSML to a much broader scope wherein Domain Specific Languages (DSL) can continue to be expressed in all their richness in a platform independent manner that is devoid of any DEVS and programming language constructs (Figure 4). The key idea being domain specialists need not delve in the DEVS world to reap the benefits of DEVS framework.

The DEVSML 2.0 stack in Figure 4 adds three transformations at the top layer:

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

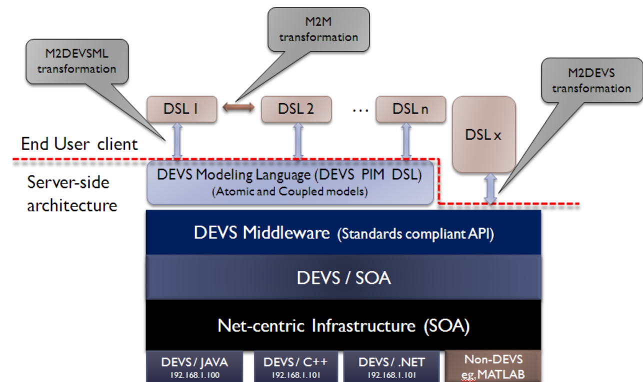

Figure 4. DEVSML 2.0 stack employing M2M and M2DEVS transformations for Model and simulator transparency

The end-user as indicated in Figure 4 will develop models in their own DSL and the DEVS expert 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. There are many DEVS DSLs that implement a subset of rigorous DEVS formalism. One example of DEVS DSL is XML-based Finite Deterministic DEVS (XFFDEVS) [21]. DEVSSpecML [22] built on BNF grammar is another example of DEVS DSL. DSLs can be created using many available tools and technologies such as: Generic Modeling Environment (GME) [23], Xtext, Ruby, Scala and many others. GME is considered as the centerpiece modeling technology for MIC.

To develop a DSL in GME, a *meta-modeler* specifies its abstract and concrete syntaxes in GME. The abstract syntax captures the concepts, constraints and relationships relevant to a domain using abstractions that exploit domain-specific knowledge and processes. The concrete syntax allows a modeler, acting more like an end user than a programmer, to visually/textually specify models that people with similar domain expertise can easily comprehend. To use a DSL, a *modeler* configures GME so

that it supports the use of the DSL and then specifies models in the DSL's concrete syntax. While DSLs created and used through GME provide a compelling structured visual tool, their use requires DSL developers and users to learn an entirely new toolset. Alternative and equally effective DSL frameworks are therefore being evaluated. Other DSL writing tools like Xtext, Ruby, etc. focusing directly on the EBNF grammar provide an alternative foundation to develop the Abstract Syntax Tree (AST) for M2M transformations. The rich integration and code generation capabilities with open source tools like Eclipse make DSL development tools such as Xtext popular in the software modeling community.

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 models that are platform independent 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. The extended DEVML 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 [18]. This hybrid M&S capability facilitates interoperability.

The next sections will describe a new DSL for the cognitive science discipline and how the semantics are then transformed to a DEVS DSL leading to an executable model.

4. DSL FOR COGNITIVE MODELING

Many cognitive scientists consider cognitive activity to be a product of an open system that interacts with the environment [24]. This perspective has motivated such cognitive scientists to study *cognitive architecture*, the invariant structural and behavioral system properties underlying cognitive activity that remain constant across time and situation.

4.1. Overview of ACT-R

The Adaptive Character of Thought-Rational (ACT-R) is a theory of human cognition in the form of a cognitive architecture and a cognitive modeling and simulation framework [25]. A key characteristic of ACT-R is that it distinguishes between declarative and procedural knowledge. Declarative knowledge represents factual information that can be retrieved and acted upon. Units of declarative knowledge are known as *chunks* in ACT-R. Structurally, a chunk consists of a chunk-type and a list of key-value pairs. Declarative knowledge chunks are stored in a long-term associative memory. Procedural knowledge represents steps of central cognitive processing. Instances of procedural knowledge are known as *productions* in ACT-R. Productions represent associations between context-constraints and actions. Productions are stored in a flat procedural memory.

The base architecture of ACT-R consists of a set of independent modules that processes a different kind of declarative knowledge. Table 1 lists the central modules in ACT-R. Transient declarative knowledge is stored in module *buffers*. Only the module maintaining a buffer or the procedural module can modify the contents of that buffer. Cognitive activity arises from interactions between a central production system (CPS) and these modules. Productions, essentially rule-action pairs, exchange information within and between buffers during these interactions.

Overall processing activity in ACT-R consists of a mixture of parallel and serial processing in and across modules. Parallel activity can occur within each of the modules. For example, retrieval requests processed by the declarative module depend on a parallel search through long-term memory for chunks matching constraints expressed in retrieval requests. Parallel processing can also occur across the modules. For example, the motor module can manipulate the hands while the vision module identifies an object in the visual field.

Module	Role in Cognition
Audio	Localizing and identifying sounds in the environment
Declarative	Storing and retrieving information in an associative memory
Goal	Tracking progress towards current goals and intentions
Imaginal	Maintaining internal representations of problems & situations
Motor	Controlling the hands
Procedural	Initiating and coordinating the behavior of all other modules
Speech	Producing speech
Vision	Identifying objects in the visual field

Table 1. The modules and buffers making up ACT-R's architectural core.

The CPS underlying ACT-R's procedural module matches productions to context (buffer contents) and uses a utility calculus to determine which matching production fires and advances the cognitive process. The resolution of ACT-R's utility calculus is limited to the *production level*. In large cognitive models, as the number of productions increases, this level of resolution leads to computational demands that impede scalability. The demands associated with production utility calculations in a flat procedural memory can be mitigated by representing and processing procedural knowledge in a hierarchical manner using extended finite state machines (EFSM). EFSMs can be used to effectively group productions into descriptions of cognitive activity at a resolution above the production level.

A DSL supporting the specification of ACT-R procedural knowledge in EFSMs that can be incorporated into the DEVSML 2.0 stack has recently been developed [26]. This DSL, known as the Research Modeling Language (RML), illustrates how cognitive modeling can benefit from a component based modeling and simulation framework based on DEVS. The abstract syntax of RML is influenced by the ACT-R cognitive architecture [25]. We show in the next section of this paper that RML can function as a component in the DEVSML 2.0 stack and yield DEVS components that can be composed into a larger simulation that includes both DEVS and non-DEVS components.

4.2. The Syntax of RML

The abstract syntax of RML includes concepts, constraints and relations that capture the behavior of the modules listed in Table 1.

4.2.1. Declarative Knowledge

The abstract syntax of RML assumes that declarative knowledge is represented as predicates capturing relationships between entities. Transient declarative knowledge resides in a working memory. Knowledge is added to working memory by: (1) environment events; (2) active attention; (3) module processes; and (4) the direct utilization of procedural knowledge.

Declarative knowledge is maintained in a semantic network. Nodes in the network represent the classes, properties, and instances constituting a body of knowledge. Nodes are connected by edges representing relations. Retrievals based on ACT-R's retrieval equations are achieved through parallel spreading activation in the semantic network. The design and performance of RML's declarative memory system is described in [27].

4.2.2. Procedural Knowledge

The abstract syntax of RML assumes that procedural knowledge is represented in *behavior models* that explicitly represent cognitive state, context, alternative courses of action, and failure. These models are formally represented as extended finite state machines (EFSMs). EFSMs are a 4-tuple:

EFSM = <S, s0, LSV, TRA>, where

S	:	set of states
s0	:	start state
LSV	:	set of locally scoped variables
TRA	:	set of transitions

A single start state must be included in the set of states (S). A number of optional stop states may be included in S. The LSV and TRA sets can be empty.

In the following descriptions of locally scoped variables and transitions, type information is included in parentheses. Definitions and a grammar formally describing these types can be found in [26].

Locally scoped variables are a 2-tuple:
LSV = <N, V>, where

N : name *(Variable_Name)*
V : value *(Variable_Value)*

LSVs maintain representations of context. For example, aspects of declarative knowledge originating in the declarative module can be maintained in LSVs over the course of cognitive activity.

Transitions are a 9-tuple:

TRA = <P, S, D, L, Pr, Cp, F, A, Ps>, where

P : priority *(Integer)*
S : source *(State_Name)*
D : destination *(State_Name)*
L : label *(String)*
Pr: pre-bindings *(Binding)*
Cp: context patters *(Pattern)*
F : functions *(Function)*
A : assertions *(Assertion)*
Ps: post-bindings *(Binding)*

Priority (P): preferences/estimates of utility that resolve conflict when more than one transition is possible from a state.

Source (S): the state from which a transition originates. A **destination** (D) is the state to which a transition leads.

Label (L): a description of the function/purpose of a transition. Labels are similar to documentation strings.

Pre-bindings (Pr): "name=value" pairs used to: (1) ensure that LSVs have a specific value (values are constants); or (2) retrieve elements from context (values are variables).

Context patterns (Cp): predicate constraints that must be met for a transition to be allowed. Patterns can be: (1) used to ensure that particular pieces of declarative knowledge are in working memory or not (predicate patterns contain only constants); or (2) used to bind elements related by predicates in working memory (predicate patterns contain variables).

Functions (F): execute calculations involving LSVs and context pattern elements. They are provided in RML because they significantly increase the representational power of state machines.

Assertions (A): predicates added to working memory after a transition has completed.

Post-bindings (Ps): name/value pairs that will add to or overwrite LSVs maintained by an EFSM.

4.3. RML Model of the Fan-Effect

The fan-effect [28] reflects the impact of knowledge complexity on human memory. As a person memorizes additional facts involving a concept, the amount of time it takes them to retrieve any one of these facts increases. A model of the fan-effect has been developed in RML in order to demonstrate how a DSL semantically anchored in DEVS can be used to model detailed cognitive activity. The exercise of specifying a RML model of the fan-effect illustrates two points: (1) a DSL with an abstract syntax employing critical aspects of a cognitive architecture like ACT-R retains the cognitive fidelity of those aspects; (2) a DSL permitting the specification of behavior at a level of organization above the production supports the development of behavioral sub-assemblies. The first point demonstrates that new modeling formalisms designed to facilitate scale and interoperability need not abstract their users from empirically important details. The second point illustrates how complexity can be managed through hierarchy.

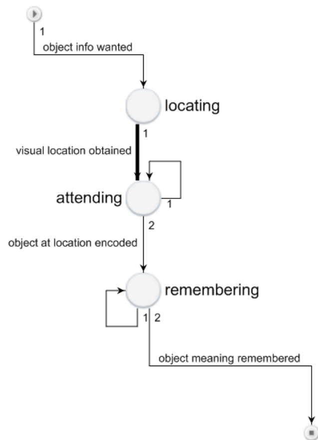

Figure 5. EFSM representing behavior that locates, attends to, and retrieves declarative knowledge about an object.

The RML fan-effect model consists of two communicating EFSMs:

- ***attend_and_comprehend***: (Figure 5) representing behavior during a repeated attend/comprehend subtask.

- ***fan_task***: (Figure 7) representing behavior at the task level.

Notice how transition labels and state names summarize and document the represented behavior at a high level of abstraction effectively concealing the formal details of the cognitive activity represented by the EFSM from the user. The formal attributes of a transition can be edited by selecting it and adding/editing textual aspects of its underlying 9-tuple (Figure 6).

Attributes | Preferences | Properties

Label	visual location obtained
PreBindings	id=Id
PostBindings	lx=Lx, ly=Ly
Patterns	{visual_location, Id, Lx, Ly}
Functions	
Assertions	{focus_attention, Lx, Ly}
Priority	1

Figure 6. Transition attributes specifying how attention is focused onto the Lx/Ly coordinates of a visual location.

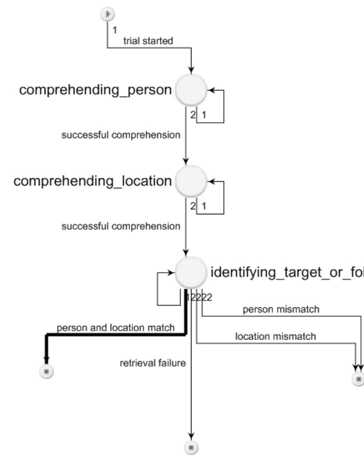

Figure 7. EFSM representing behavior that comprehends a person, comprehends a location, and then identifies a trial as a target or a foil.

The EFSM of ***fan_task*** is show in Figure 7. The states *comprehending_person* and *comprehending_location* 'task' the ***attend_comprehend*** EFSM in Figure 5. When the ***attend_comprehend*** EFSM transitions to its stop state, execution control returns to the ***fan_task*** EFSM. A transition in RML is semantically equivalent to an ACT-R production [26]. The major outcome of this exercise is that the existing productions now have been regrouped (as *transitions* in RML) based on the abstract states defined in EFSMs. This has a far reaching benefit when scalability is concerned. Instead of a flat pool of *productions* that are evaluated each simulation cycle now only a subset of *productions* are evaluated for execution of the system. In addition, now we have a behavioral unit such as ***attend_and_comprehend*** that can be reused and participate in hierarchical construction of other EFSMs.

The transition process in RML EFSMs during simulation is based on: (1) the accumulation of *match bindings* when *patterns* match *context* (knowledge in working memory); (2) the optional augmentation of *match*

bindings through *functions*; and (3) the instantiation of *assertions* with *match bindings*. As transitions occur during runtime, a sequential pattern matching process realizes a type of forward chaining. If a transition time cost of 50ms and ACT-R's time costs for attention shifts, motor responses, and declarative retrievals are adopted during simulation, this forward chaining precisely matches production firing in ACT-R. When the RML model of the fan-effect is simulated in either the Erlang-based Run time Environment [26] or the DEVSML 2.0 stack as described ahead, retrieval successes, failures, latencies, and task actions precisely matching those of a fan-effect model described in ACT-R instructional materials are produced.

5. TRANSFORMING RML TO DEVSML

In the previous section we saw how a DSL such as RML can specify the similar behavior of an ACT-R fan-effect model. We also mentioned that the RML is semantically anchored in Erlang that allows execution of an RML model. While the objective of moving away from ACT-R production system through a better abstraction mechanism has been achieved, it lands us in the same situation wherein isolated simulation platforms fail to be a component in a larger system of system. In order to incorporate RML models (in their current state) in an integrated simulation exercise, we have to also consider the execution platform interoperability as well, not to mention the pragmatic, semantic and syntactic interoperability [29].

The approach proposed in this paper takes the RML meta-model in its entirety and decouples it from its semantic anchoring with Erlang. Once that decoupling is done, the meta-model is then semantically anchored in DEVS, which provides solutions to interoperability, extensibility, composability and scalability. It can be argued that we are semantically anchoring RML in just another platform that provides some additional benefits. We would like to take the position that DEVS is more than a computational platform. It is a formalism and a complete framework supported by dynamical systems theory that has transparent platform execution with its SOA implementation [11,30]. Figure 2 has already emphasized this point. We will now show how RML is semantically anchored in DEVS and a M2DEVSML transformation is performed.

5.1. From RML elements to DEVS components

From structure perspective, any DEVS system is made up of three elements, the model components (atomic or coupled), the messages that flow between them, and the couplings that communicate these messages between components. Both the atomic and coupled DEVS components transmit and receive messages. However, the capacity to interpret the message and use it to express the behavior is solely the characteristic of DEVS atomic component. A new message originates exclusively within an atomic component per its behavior specification and is then placed at the output interface of the atomic component. The behavior of an atomic component is a function of superposition of two behaviors i.e. when an external message is received and when it is not. In order to specify the behavior, a *state space* is specified and the transitions between these states are defined with respect to an 'event' abstraction. An *Event* can occur in only two modes i.e. when an external *message is received* or an internal variable goes through a value change so as to mark an 'event'. Each designed event is then associated with a state within the *state space* and the transition to the next state is performed when such 'event' occurs. In DEVS, this transition is specified in two functions i.e. the *external transition function* and the *internal transition function*. Further, after every such state transition, whether external or internal, if the component needs to send an *ouput message in that specific state*, an *output function* is specified that puts the message on output interface of this atomic component. Formally speaking the atomic DEVS is defined as an 8 tuple system:

$$M = < X; S; Y; \delta_{int}; \delta_{ext}; \delta_{con}; \lambda; ta >$$

where,
X is the set of input values
S is the set of states
Y is the set of output values
$\delta_{int} : S \rightarrow S$ is the internal transition function
$\delta_{ext} : Q \times X_b \rightarrow S$ is the external transition function;
where X_b is a set of bags over elements in X;
Q is the total state set
$\delta_{con} : S \times X_b \rightarrow S$ is the confuent transition function,
subject to $\delta_{con}(s; \Phi) = \delta_{int}(s)$
$\lambda: S \rightarrow Y_b$ is the output function
$ta : S \rightarrow R_{(0+;inf)}$ is the time advance function

Describing the richness of DEVS atomic behavior is outside the scope of this paper. We will consider a subset of DEVS called XML-Based Finite Deterministic DEVS (XFDDEVS) [21] that abstracts the DEVS formalism to an automated transformation process using XML such that true DEVS semantics is maintained. We can consider FDDEVS to be another DSL that is semantically anchored in DEVS.

The RML state machine in Figure 5 is a state machine with finite number of states. The atomic FDDEVS is specified as a seven tuple: $<X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta>$, where **X** is set of incoming messages, **Y** is a set of outgoing messages, **S** is the set of states, $\boldsymbol{\delta_{int}}$ is the internal transition function, $\boldsymbol{\delta_{ext}}$ is the external transition function, $\boldsymbol{\lambda}$ is the output function and **ta** is the timeout for each state.

We will now transform the RML description for EFSM ***attend_and_comprehend*** as shown in Figure 5. As we already have 3 states i.e. *locating*, *attending* and

remembering, we have an initial starting point for our state space. However, we have to remember that the notion of 'state' in DEVS is associated with occurrence of an 'event'. Now, looking at each of the transitions in each of the three states in Figure 5, we find that each transition although specifies the source *src* and the destination *dst* state, has more going on inside it. For example, the *pre_binds*, *post_binds*, *patterns* and *assertions* elements. As per the RML semantics, the model will expect the *pre_bind* variables to match up with the *patterns*, and if matched, will perform the *post_bindings* and *assertions* and will then finally move to the *dst* state. In DEVS semantics, this operation can be considered as two events, and consequently, two states. The first state being, *beginOperation*, wherein evaluation is being made per input *patterns* and the second state being, *dst* itself. On completion of first state, *assertions* (output) is being sent and the model then moves to *dst* state. While there is no problem in the RML semantics, the DEVS formalism requires the specification of *output function* which is associated with a specific state. If we preserve the RML state set then the point where two events happen together, ie. *Incoming patterns* and *assertions*, breaks the notion of discrete event in DEVS formalism. The DEVS semantics very clearly expresses this in the *output function*. Using the system homomorphism concepts [1] as shown in Figure 8, by introducing a Zero time state (Figure 9), we not only preserve the RML semantics but also transform the state machine into a DEVS state machine.

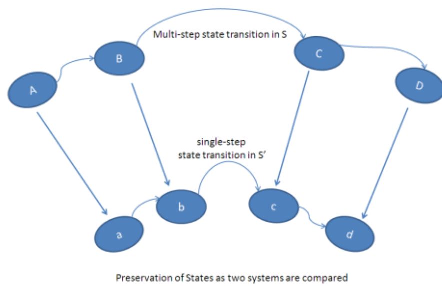

Figure 8. Preservation of States as two systems are compared and M2DEVS transformation is performed

The automated transformation is executed using Xtext, XPand and XML-based technologies such as JAXB. Xtext is a language development framework that allows development of rich DSLs backed by EBNF grammar in a full blown auto-generated Eclipse editor with interpreters and compilers. It allows syntax highlighting, code completion, code validation, quick fixes and complete file operation capabilities within the Eclipse IDE. Xpand is the code generation technology built on top of Xtext that allows creation of code-templates that uses the underlying EBNF in the designed DSL for its further use and transformation. The utility of Xpand is like XSLT in the XML domain wherein XSLT acts as the transformer between two markup languages. In our example, the EBNF for RML has been discussed in [26]. Using Xtext and Xpand, the EBNF elements are extracted and transformed directly to XFDDEVS as per rules defined in Table 2. An Eclipse plugin was created that performed the complete process i.e. RML-to-FDDEVS-to-DEVSJAVA. More details are reserved for our extended article.

RML Elements		FDDEVS Elements
Globals		
	states	S
Transition		If *patterns* >0, then each *tuple* in *patterns* is an incoming external message and be addressed in ***δext***. The *src* state must transition to *beginDst* state in zero time. if *assertions*>0 then each tuple is an outgoing message and be addressed in **λ** in state *beginDst* every *beginDst* state should internally transition to *dst* in 0ms . Every dst must match the ta=50ms of RML state and once elapsed should internally transition to passive.
	src	s in S
	dst	s in S
	patterns	X
	assertions	Y

Table 2. Semantic mapping from RML to FDDEVS

Figure 9 shows the complete DEVS state machine after this transformation process. The solid lines shows external event i.e. incoming message depicted with prefix **?**. The dotted lines show internal event transitions. The generated messages are depicted with a prefix **!**. The timeout for each state are in the parenthesis. Table 3 shows the mapping of states in the RML EFSM (Figure 5) to those in the FDDEVS State Machine (Figure 9) along with timing in parenthesis. Table 3 lists the mapping of RML semantics into XFDDEVS elements.

RML States	FDDEVS States
start (0)	passive(infinity)
locating(50)	beginLocating(0), locating(50)
attending(50)	beginAttending(0), attending(50)
remembering(50)	beginRemembering(0), remembering(50)
stop(0)	passive(infinity)

Table 3. RML states to FDDEVS states

Similarly, the second RML Fan Task EFSM as depicted in Figure 7 is also transformed to FDDEVS. Finally, the two atomic models are coupled and realized in DEVSJAVA simulation viewer [10] as shown in Figure 10. The Fan Task

atomic *tasks* the *attend_and_comprehend* atomic by sending 'assert_intention' message from 'out_assert_intention' outport to 'in_assert_intention' inport. This coupling is specified explicitly unlike the Erlang Runtime Environment where communications between the EFSMs are handled through'blackboard' architecture. This is another advantage of DEVS component based architecture wherein components like blackboard which are not scalable in a distributed environment can be eliminated in their entirety. Further, such explicit coupling facilitates standardization of interfaces for the developed components.

We have shown the execution of M2DEVSML transformation from one DSL into another DSL that is semantically anchored in DEVS. We have addressed the atomic behavior, coupling and structure of the transformed RML model into DEVS atomic and coupled models. The remaining piece of the entire transformation exercise is the message transformation which will be reported in an extended article.

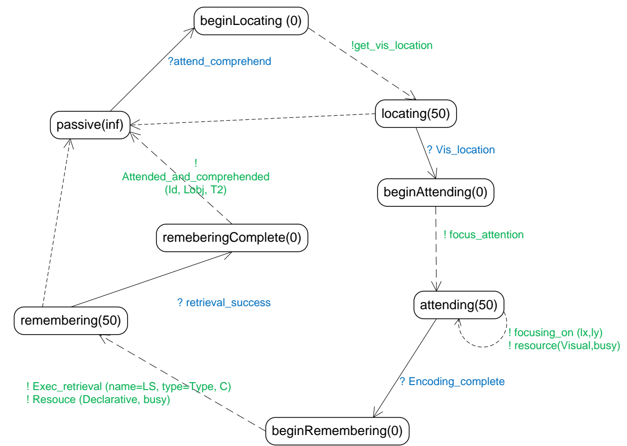

Figure 9. FDDEVS state machine for *attend_and_comprehend*

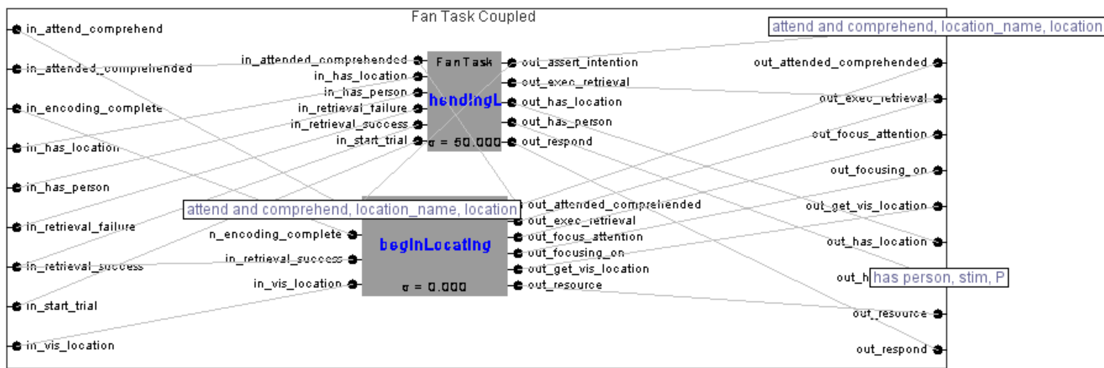

Figure 10. Fan Task Coupled Model

6. CONCLUSIONS AND FUTURE WORK

AFRL research efforts employing cognitive modeling are growing in scope. These efforts to transition cognitive modeling from the laboratory to operations settings are struggling to meet challenges associated with: (1) increasing the scale of models; and (2) integrating models into software-intensive distributed task environments. An AFRL LSCM initiative is researching solutions to these challenges based on high-level languages, more specifically the DSLs, for describing cognitive models at a higher level of abstraction to facilitate scale and the underlying simulation frameworks supporting these DSLs. DEVS formalism and M&S framework allow users to simulate models in architectures that improve model integration and interoperability in a net-centric domain using the DEVSML stack 2.0. This paper described RML, a hybrid (textual/visual) cognitive and behavior modeling DSL influenced by ACT-R in which models capturing cognitive activity above the level of the production can be specified. RML illustrated how DSLs designed to facilitate scale and interoperability need not isolate users from empirically important details.

We also extended the earlier DEVSML stack with DSLs and suggested M2M, M2DEVSML and M2DEVS transformations as the preferred way to achieve model interoperability and larger integration of modeling framework with an underlying DEVS distributed simulation infrastructure. We illustrated this concept by developing a DSL called RML for ACT-R and executing it on DEVS platform.

Current efforts to make net-centric cognitive models will allow cognitive modelers to evaluate and field their models through Service Oriented Architectures (SOA) and other net-centric infrastructures. This paper lays the foundation and suggests how future work will amplify the benefits of componentizing other DSLs from disparate domains. By unifying ACT-R modeling practices into the DEVS Unified Process [31], future versions of DEVS/ACT-

R will facilitate the verification and validation of ACT-R models. Refinements and extensions to net-centric DEVS/ACT-R will enable cognitive scientists to “black-box” models of cognitive activity into larger system of systems [32].

The work described in this paper illustrates how methods and processes common in general M&S can be exploited by other fields—in this case cognitive modeling. The immediate contribution of this work is platform independent modeling and simulation using DSLs and DEVSML stack 2.0, an architecture that is allowing AFRL to begin integrating cognitive models into net-centric infrastructures such as a Service Oriented Architecture (SOA). The transition of DEVS/ACT-R to a SOA has the potential to literally revolutionize how AFRL develops and fields large-scale cognitive models.

References

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

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

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

[4] Sztipanovits, J., & Karsai, G., “Model-integrated computing. *Computer* “, *30* (4), 110-111. 1997

[5] Balasubramanian, K., Schmidt, D. C., Molnár, Z., & Lédeczi, A. “Component-Based System Integration via (Meta)Model Composition.” *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (pp. 93-102). Washington, DC: IEEE Computer Society, 2008

[6] Balasubramanian, K., Schmidt, D. C., Molnár, Z., & Lédeczi, A. “System Integration using Model-Driven Engineering”. In P. F. Tiako, *Designing Software-intensive Systems: Methods and Principles* (pp. 474-504). Idea Group Inc., 2008

[7] 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.

[8] 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.

[9] 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

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

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

[12] Fujimoto, RM, "Parallel and Distribution Simulation Systems", Wiley, 1999

[13] Seo, C, Park, S, Kim, B, Cheon, S, Zeigler, BP, "Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment", Advanced Simulation Technologies conference (ASTC), Arlington, VA, 2004

[14] Cheon, S, Seo, S, Park, S, Zeigler, BP, "Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Networked System", Advanced Simulation Technologies Conference, Arlington, VA, 2004

[15] Kim, K, Kang, W, "CORBA-Based Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One", International Conference on Computational Science and Its Applications, Italy 2004

[16] Zhang, M, Zeigler, BP, Hammonds, P, "DEVS/RMI-An Auto-Adaptive and Recongurable Distributed Simulation Environment for Engineering Studies", ITEA Journal, July 2005

[17] 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

[18] 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)

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

[20] Xpand Model Transformation Framework accessible at: http://www.eclipse.org/modeling/m2t/?project=xpand

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

[22] 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

[23] GME: Generic Modeling Environment, last accessed Jan 2011 at: http://www.isis.vanderbilt.edu/Projects/gme/

[24] Wilson, M. (2002). Six views of embodied cognition. *Psychonomic Bulletin & Review , 9* (4), 625-636.

[25] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S. A., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. Psychological Review , 111 (4), 1036-1060.

[26] Douglass, SA, Mittal, S, “Using Domain Specific Languages to Improve Scale and Integration of Cognitive Models”, Behavior Representation in Modeling and Simulation, Utah, USA, March 2011

[27] Douglass, SA & Myers, CW, “Concurrent knowledge activation calculation in large declarative memories”. In D. D. Salvucci & G. Gunzelmann (Eds.), Proceedings of the 10th *International Conference of Cognitive Modeling*, Philadelphia, Pennsylvania, USA, 2010

[28] Anderson, J.R. & Reder, L.M, “The fan effect: New results and new theories”. *Journal of Experimental Psychology: General, 128(2)*, 186-197, 1999

[29] 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

[30] Mittal, S, "Agile Net-centric System using DEVS Unified Process", chapter for "Intelligence Based Systems Engineer", Ed. Andreas Tolk, Lakhmi Jain, Springer-Verlag 2011

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

[32] Mittal, S, Douglass, SA, “Net-Centric ACT-R Based Cogntive Architecture with DEVS Unified Process”, DEVS Symposium, Spring Simulation Multiconference, Boston, April 2011