Using Domain-Specific Languages to Improve the Scale and Integration of Cognitive Models

Scott A. Douglass
Air Force Research Laboratory
6030 S. Kent St.
Mesa, AZ, 85212
480-988-6561 x675
scott.douglass@mesa.afmc.af.mil

Saurabh Mittal
L-3 Communications, Air Force Research Laboratory
6030 S. Kent St.
Mesa, AZ, 85212
480-988-6561 x677
saurabh.mittal@L3com.com

Keywords:
Domain-Specific Languages, ACT-R, Cognitive Modeling and Simulation, DEVS

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 cognitive modeling and simulation framework that exploits the Discrete Event System Specification (DEVS) formalism to eliminate these impediments. Domain specific languages (DSLs) used in the framework facilitate model scale and interoperability. The framework and an example DSL called research modeling language (RML) will be discussed.*

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 training environments. The struggle has two sources: (1) the need to specify detailed knowledge and process descriptions in our modeling frameworks; (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) research initiative is developing solutions to these scale and interoperability challenges based on high-level languages for describing cognitive models and simulation frameworks supporting them based on the Discrete Event System Specification (DEVS) formalism (Zeigler, Kim & Praehofer, 2000). This paper discusses a cognitive modeling and simulation framework that represents our best solution so far. The paper begins with an overview of the approaches and objectives of LSCM. Then the paper describes the actual framework. The remainder of the paper describes the research modeling language (RML), one of the domain-specific languages (DSLs) supported in the framework. This discussion of RML demonstrates how cognitive models can be specified in DSLs that facilitate scale through abstraction. The discussion also explains how RML models are executed in the cognitive modeling and simulation framework.

2. Large Scale Cognitive Modeling (LSCM)

The LSCM initiative is seeking to adapt and exploit methods and practices from Model Integrated Computing (MIC) and advanced modeling and simulation (M&S) in order to help cognitive modelers increase the scale and interoperability of their models:

1. MIC is being adapted and exploited to facilitate DSL development and model/systems integration.
2. Advanced M&S is being adapted and exploited to achieve increased scale and interoperability.

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 (Sztipanovits & Karsai, 1997). 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 (Balasubramanian, Schmidt, Molnár, & Lédeczi, 2007): (3) automates the integration of models specified in these DSLs into task environments or larger systems (Balasubramanian, Schmidt, Molnár, & Lédeczi, 2008); and (4) provides automated model-to-model (M2M) transformation capabilities that produce executable code artifacts from models specified in these DSLs.

2.2 Modeling and Simulation (M&S) using DEVS

The LSCM initiative is also exploring how the Discrete Event System Specification (DEVS) formalism (Zeigler, Kim & Praehofer, 2000) can be used to semantically anchor DSLs. This aspect of LSCM is investigating how computationally realizing DSLs in DEVS: (1) provides them with a behavioral semantics that can be directly executed in advanced DEVS simulators; and (2) allows models specified in them to interoperate with other DEVS and HLA compatible components in broader systems-of-systems (Zeigler, Mittal, & Hu, 2008).

3. A Component-Based Cognitive Modeling and Simulation Framework

As our cognitive 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 cognitive modeling. 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 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. The architecture technically realizes a Discrete Event System Specification Modeling Language (Mittal, Martin & Zeigler, 2007) in a DEVSML stack. 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.

3.1 DEVS Modeling Language (DEVSML) Stack

An earlier DEVSML stack realized models in Java and in a platform independent DEVS Modeling Language that used XML as a means of message passing (Mittal, Martin & Zeigler, 2007). The model semantics were bound together by XML and JAVAML was used to translate Java models into the XML. DEVSML is based on an EBNF grammar and is supported by a DEVS standards-compliant middleware API. The middleware enables model execution in a net-centric service oriented architecture (SOA).

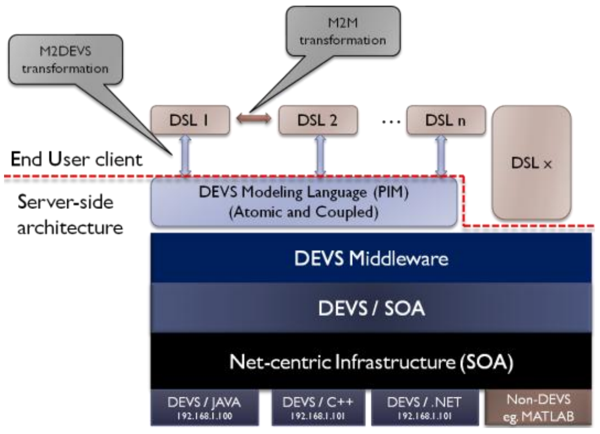


Figure 1. Extended DEVSML stack using Model-to-Model (M2M) and Model-to-DEVS (M2DEVS) transformations to enhance model and simulator transparency.

Research efforts in LSCM are extending the current DEVSML stack so that DSLs can be used to specify platform independent models (PIMs) that are devoid of any DEVS and programming language constructs (Figure 1). The two pieces that have been added to the current DEVSML stack to enable the use of DSLs are the Model-to-Model (M2M) transformation and the Model-to-DEVS (M2DEVS) transformation. To capitalize on these new pieces, the user develops models in DSLs and employs M2M and/or M2DEVS transformations to provide a DEVS execution "backend" to the models. The key benefit of these additions is that domain specialists need not develop DEVS expertise to use the DEVSML stack.

3.2 Domain-Specific Languages (DSLs)

DSLs used in the DEVSML stack are developed using the Generic Modeling Environment (GME), the centerpiece modeling technology of MIC. To develop a DSL, 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.

DSLs developed in GME can be formally related to each other by integrating their abstract syntaxes. DSLs related in this way can undergo automated model-to-model (M2M) transformations. These M2M transformations can translate a model specified in one DSL into a model expressed in another related DSL. M2M transformations can also relate DSLs to languages that execute in simulators. Model-to-DEVS (M2DEVS) transformations available in the extended DEVSML stack (Figure 1) translate models specified in DSLs into DEVSML. Since DEVSML interoperates with DEVS middleware, the M2DEVS transformation process semantically anchors DSLs in DEVS and makes them executable. M2M transformation can be performed in GME and/or in technologies such as Xtext/Xpand (see references for information).

3.3 Benefits of an Extended DEVSML Stack

The addition of M2M and M2DEVS transformations to the DEVSML stack adds true model and simulator transparency to a net-centric SOA infrastructure. The transformations yield DEVS models that are platform independent models (PIMs) that can be developed, compared and shared in a collaborative process. Finally, 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. This hybrid M&S capability facilitates interoperability.

4. Research Modeling Language (RML)

The research modeling language (RML) is a DSL used to specify cognitive models in the DEVSML stack. The abstract syntax of RML is influenced by the ACT-R cognitive architecture (Anderson, 2007). The concrete syntax of RML is designed so that a modeler with experience in ACT-R can specify behaviorally equivalent models at a higher level of abstraction.

4.1 The Abstract Syntax of RML

The abstract syntax of RML includes concepts, constraints and relations that capture the behavior of a set of independent modules that each processes a different kind of knowledge. Cognitive activity arises from interactions between behavioral representations and these modules.

Module	Role in Cognition
Audio	Localizing and identifying sounds in the environment
Declarative	Storing and retrieving information in an associative memory
Motor	Controlling the hands
Speech	Producing speech
Vision	Identifying objects in the visual field

Table 1. Modules assumed in RML's abstract syntax.

4.1.1 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. Nodes maintain information about: (a) retrieval parameters; (b) reference histories; and (c) last activation levels. Nodes use ACT-R's chunk activation equations to compute their activations and therefore mimic ACT-R's frequency, recency and memory decay effects. New nodes are acquired and existing nodes strengthened in such a way that declarative learning in the semantic network replicates the behavior of ACT-R's declarative memory. Retrievals are achieved through ACT-R's retrieval equations and parallel spreading activation in the semantic network. Douglass & Myers (2010) describe the design and performance of RML's declarative memory system.

4.1.2 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. *There is nothing corresponding to EFSMs in ACT-R; it is not possible to explicitly represent behavior organized above the level of the production in ACT-R.*

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 Appendix A.

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. *LSVs maintain context in the same way key/value pairs represent context in ACT-R buffer chunks.*

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 patterns *(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. *Source and destination indicators are similar to state-specific key/value pairs used in ACT-R models to maintain behavior across productions.*

Label (L): a description of the function/purpose of a transition. *Labels are similar to documentation strings that can be associated with ACT-R productions.*

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). *Pre-bindings are similar to left-hand-side key/value constraints in ACT-R.*

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). *Context patterns are similar to left-hand-side (LHS) key/value constraints in ACT-R.*

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. *Assertions are similar to right-hand-side (RHS) key/value actions in ACT-R.*

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

4.2 The Concrete Syntax of RML

RML’s concrete syntax provides users with a hybrid (visual/textual) language in which they specify the declarative and procedural knowledge underlying a model. RML’s meta-model includes constraints that check the validity of models. These constraints guide modeler actions and ensure that the concrete syntax of a RML model is *correct by construction.*

4.2.1 Declarative Knowledge

As previously stated, RML’s abstract syntax assumes that declarative knowledge is represented as predicates capturing binary relationships between entities. This assumption allows RML to accommodate declarative knowledge specified in any OWL-compatible ontology authoring application. Declarative knowledge can currently be specified in GME or Protégé. Douglass & Myers (2010) describe the role ontologies play in RML and give visual/textual examples of declarative knowledge.

4.2.2 Procedural Knowledge

RML’s concrete syntax allows modelers to visually/textually specify procedural knowledge (EFSMs) in GME. EFSMs are individually specified and can be grouped into libraries that facilitate the development of behavior model repositories.

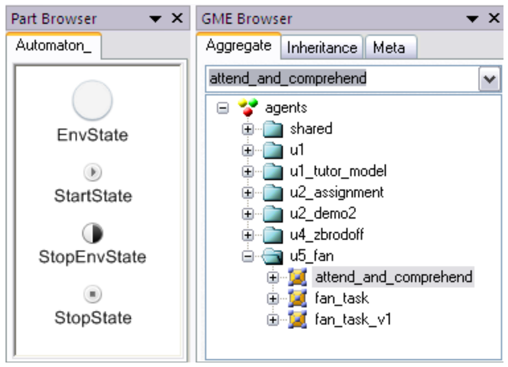


Figure 2. Visual aspects of RML’s concrete syntax. The *Part Browser* provides users with a pallet of states and the *GME Browser* helps them to organize behavior models into libraries.

The individual EFSM corresponding to the ***attend_and_comprehend*** aggregate selected in Figure 2 is shown in Figure 3. 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 EFSM from the user.

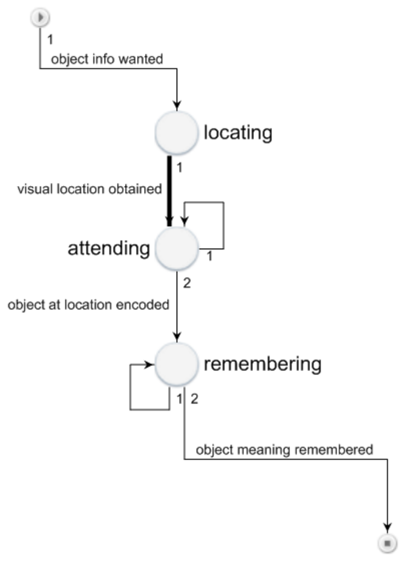
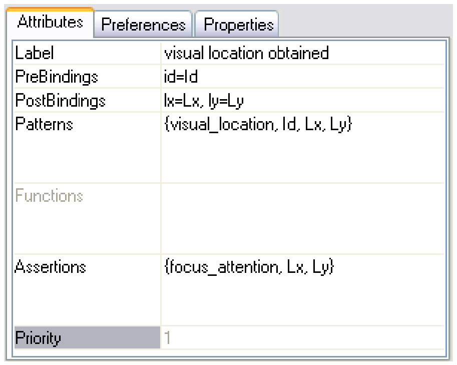


Figure 3. EFSM representing behavior that locates, attends to, and retrieves declarative knowledge about an object.

States are added to an EFSM by selecting one of the desired types from GME's *Part Browser* and clicking where in the EFSM the new state should be positioned. Transitions are added to an EFSM by clicking the source state and then dragging a *connection* to the destination state. The formal attributes of a transition can be edited by selecting it and adding/editing textual aspects of its underlying 9-tuple.



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 4. Transition attributes specifying how attention is focused onto the Lx/Ly coordinates of a visual location. Note how the source (S) and destination (D) attributes are missing. They are explicit in EFSM diagrams and therefore not textual transition attributes.

4.3 The Semantic Anchoring of RML

Models specified in RML can obtain behavioral semantics through two transformation processes. Each process semantically anchors RML in a M&S framework that supports model execution and performance logging.

4.3.1. Transforming RML to Erlang

To explore how concurrency in computer languages and multi-core CPUs facilitate scale, we have developed a RML translation and runtime environment (RTE) in the Erlang programming language (Cesarini & Thompson, 2009). The RTE automatically translates OWL-compatible ontologies (declarative knowledge) and EFSMs authored in GME (procedural knowledge) into executable Erlang. Ontologies are translated into node and edge descriptions that are used by the RTE to configure a semantic network. EFSMs are transformed into executable Erlang modules.

4.3.2. Transforming RML to DEVS

To explore how a component-based M&S framework facilitates scale and interoperability, we have also developed a M2DEVS transformation that anchors RML in the DEVSML-based framework. The transformation is based on a subset of DEVS called XML-Based Finite Deterministic DEVS (XFDDEVS) (Mittal, Zeigler, Ho 2008). XFDDEVS is essentially a DSL that is semantically anchored in DEVS. XFDDEVS allows users to specify finite-deterministic state machines. These specifications can then be automatically transformed into the DEVS formalism. To transform RML EFSMs into DEVS the M2DEVS translation process: (1) transforms RML into XFDDEVS; and (2) transforms XFDVEVS into DEVS.

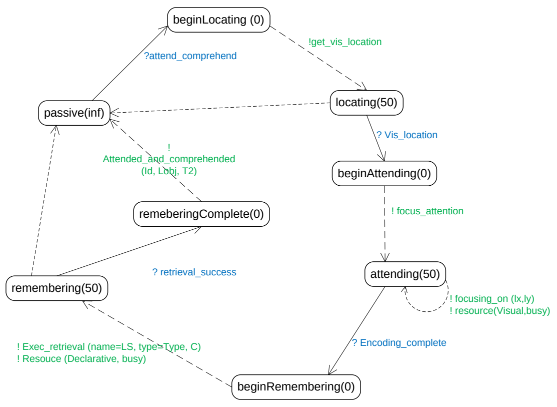


Figure 5: FDDEVS state machine corresponding to ***attend_and_comprehend.*** The state labeled "passive" corresponds to the start state of the EFSM.

Figure 5 shows a FDDEVS state machine after the automated transformation process. The solid lines show external events i.e. incoming messages 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. The FDDEVS state machine in Figure 5 computationally realizes the RML EFSM in Figure 3 in the DEVSML stack and is executed in simulation.

5. RML Model of the Fan-Effect

The fan-effect (Anderson & Reder, 1999) 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 its application in cognitive modeling.

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. Models can be built from sub-assemblies that conceal complexity rather than large numbers of primitives that expose complexity.

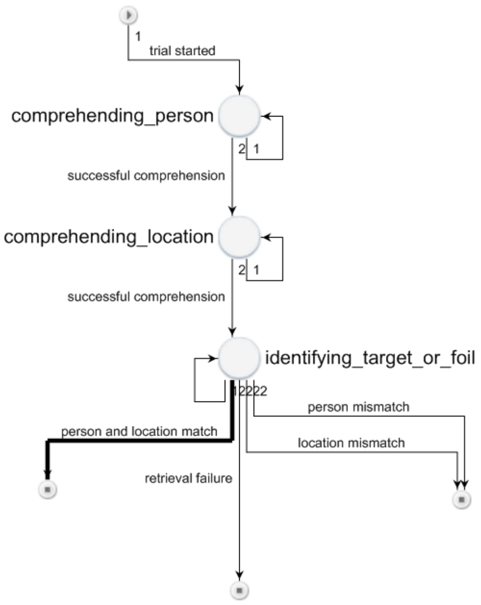


Figure 6. EFSM representing behavior that comprehends a person, comprehends a location, and then identifies a trial as a target or a foil.

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

- ***attend_and_comprehend***: (Figure 3) representing behavior during a repeated attend/comprehend subtask.
- ***fan_task***: (Figure 6) representing behavior at the task level.

While a complete description of the RML model is beyond the scope of this paper, a description of a single transition will illustrate the similarity between RML transitions and ACT-R productions. The top cell of Figure 7 shows the attributes of the transition selected (bold) in Figure 6. These attributes are expressed in a transition-centric textual DSL developed in Xtext. The DSL provides users with an alternative way to author/edit RML EFSMs. The bottom cell of Figure 7 shows a comparable production from an ACT-R model of the fan-effect.

```
transition {
    priority   2
    label      "person and location match"
    src        identifying_target_or_foil
    dst        stopstate
    pre_binds  p=P,l=L
    patterns   {retrieval_success, C, _},
               {has_person, C, P},
               {has_location, C, L}
    assertions {respond, {press_key, "k"}}
}
```

```
(P yes
    =imaginal>
       ISA     comprehend-sentence
       arg1    =p
       arg2    =l
    =retrieval>
       ISA     comprehend-sentence
       arg1    =p
       arg2    =l
    ?manual>
       state   free
==>
    +manual>
       ISA     press-key
       key     "k"
)
```

Figure 7. RML EFSM transition *"person and location match"* compared to an equivalent ACT-R production.

The RML transition explicitly represents source (src) and destination (dst) states. The ACT-R production lacks explicit state constraints and therefore either: (a) depends on implicit state constraints; or (b) represents key press behavior in any context where matching chunks of type *comprehend-sentence* are available in the imaginal and retrieval buffers. The RML transition uses pre-bindings *P* and *L* to ensure that binary relations *has_person* and *has_location* relate *C* (an event) to *P* (a person) and *L* (a location). The ACT-R production uses variables *=p* and *=l* to similarly ensure that the retrieved *comprehend-sentence* conforms with *arg1* (a person) and *arg2* (a location) represented in the imaginal buffer. The RML transition and the ACT-R production have functionally equivalent assertions and right-hand-side actions.

The EFSM transition process in the RML RTE 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 closely matches production firing in ACT-R. When the RML model of the fan-effect is simulated in either the Erlang RTE or the DEVSML stack, retrieval successes, failures, latencies, and task actions ***precisely matching*** those of the fan-effect model described in the ACT-R instructional materials are produced.

6. Conclusions

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 task environments. An AFRL LSCM initiative is researching solutions to these challenges based on high-level languages (DSLs) for describing cognitive models and simulation frameworks supporting them. These DSLs allow users to specify models in formalisms that use abstractions and re-useable sub-assemblies to achieve scale. The M&S frameworks allow users to simulate models in architecture that improves model integration and interoperability using the DEVSML stack.

This paper describes RML, a DSL influenced by ACT-R in which cognitive activity can be explicitly represented above the level of the production. RML illustrates how DSLs designed to facilitate scale need not isolate users from empirically important details. In introducing RML, we faced a choice of either showing how organizing behavior models above the level of the production facilitates scale or demonstrating how models executing in the DEVSML stack can be as cognitively plausible as an ACT-R model. We opted for the latter choice and demonstrated, with a model of the fan effect, that RML does not abstract users from empirically important details. On-going and future research efforts will demonstrate the usability and scalability of RML and other LSCM initiative DSLs.

7. References

Anderson, J.R. (2007). *How Can the Human Mind Occur in the Physical Universe?* Oxford: OUP.

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

Balasubramanian, K., Schmidt, D.C., Molnár, Z. & Lédeczi, A. (2007). 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.

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

Cesarini, F., & Thompson, S. (2009). *Erlang Programming.* O'Reilly Media, Inc.

Douglass, S.A., Ball, J. & Rogers, S. (2009). Large declarative memories in ACT-R. In *Proceedings of the 9th International Conference of Cognitive Modeling*, Manchester, United Kingdom.

Douglass, S.A. & Myers, C.W. (2010). 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.

Mittal, S., Martin, J.L.R. & Zeigler, B.P. (2007). DEVSML: Automating DEVS simulation over SOA using transparent simulators, In *Proceedings of DEVS Syposium*.

Mittal, S., Zeigler, B.P., Ho, M.H (2008), XFDDEVS: XML-Based Finite Deterministic DEVS at: *http://www.duniptechnologies.com/research/xfddevs/*

Molnár, Z., Balasubramanian, D. & Lédeczi, A. (2007). An introduction to the Generic Modeling Environment. *In Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum.* Zurich.

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

Xpand Model Transformation Framework accessible at: *http://www.eclipse.org/modeling/m2t*

Xtext Language Development Framework accessible at: *http://www.eclipse.org/Xtext/*

Zeigler, B.P., Mittal, S. & Hu, X. (2008). Towards a formal standard for interoperability in M&S/system of systems integration. *Proc. GMU-AFCEA Symposium on Critical Issues in C4I.*

Zeigler, B.P., Praehofer, H. & Kim, T.G. (2000). *Theory of Modeling and Simulation* (2nd Edition). Academic Press.

8. Acknowledgements

This research was funded through AFOSR grant #10RH05COR.

9. Author Biographies

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.

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, cognitive systems, and multiplatform modeling.

Appendix A

Atoms are constant literals that stand for themselves. They start with a lowercase letter. Subsequent characters can be uppercase, lowercase, numbers, or '_'.

Variables are used to store the value of simple or composite data types. They start with an uppercase letter. Subsequent characters can be uppercase, lowercase, numbers, or '_'.

Tuples are a composite data type. They are used to store collections of items. These items need not be the same type. Tuples are delimited by "{" and "}". Elements in a tuple are separated by ",".

Lists are a composite data type. They are used to store collections of items. List items need not be the same type. Lists are delimited by "[" and "]". Elements in a list are separated by ",". Lists can be broken into a head and a tail with a constructor operator "|".

Additional aspects of RML EFSMs are defined in the following grammar.

Note:
A : Exactly 1 A
A? : 0 or 1 A
A+ : 1 or more A
A* : 0 or more A

Base Types

```
Number          ->  (Integer | Float)
Atomic          ->  (Atom | Variable | String | Number | [] | _)
Composite       ->  (List | Tuple)
```

States and Variables

```
State_Name      ->  Atom
Variable_Name   ->  Atom
Variable_Value  ->  Member
```

Transitions

```
Binding         ->  Variable_Name '=' Member
Pattern         ->  Tuple ('=' Variable_Name)* Guard*
Function        ->  (Atomic | Composite) '=' (Case | Comp_Exp | Arith_Exp | Funcall | Atomic | Composite)
                     | Logic_Exp
Assertion       ->  Tuple
```

Expessions

```
Arith_Literal   ->  (Funcall | Number | Symbol | Variable)
Arith_Exp       ->  Expression using arithmetic operators, optional parentheses and Arith_Literals
Comp_Literal    ->  (Arith_Exp | Atomic | Composite)
Comp_Exp        ->  Expression using comparison operators, optional parentheses and Comp_Literals
Logic_Literal   ->  (Comp_Exp | Arith_Exp)
Logic_Exp       ->  Expression using logical operators, optional parentheses and Logic_Literals
```

Miscellaneous

```
Funcall         ->  Function '(' (Member (',' Member)*)? ')'
Member          ->  (Logic_Exp | Funcall | Atomic | Composite)
Guard           ->  'when' Logic_Exp ;
CaseTest        ->  (Atomic | Composite) Guard? ;
CaseRes         ->  (Case | Atomic | Composite | Logic_Exp) ;
Case            ->  'case' (Logic_Exp | Funcall | Var_Name | Atomic | Composite) 'of'
                         ((CaseTest '->' CaseRes+) (';' (CaseTest '->' CaseRes+))*)?
                     'end'
```