Visual and Persistence Behavior Modeling for DEVS in CoSMoS

Mostafa D. Fard
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Email: M.Derakhshandeh@ut.ac.ir

Hessam S. Sarjoughian
Arizona Center for Integrative Modeling &Simulation
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
Email: Sarjoughian@asu.edu

ABSTRACT
An integrated visual modeling and simulation tool called Component-based System Modeling and Simulation (CoSMoS) is extended to support behavioral specification of parallel atomic DEVS model. An approach based on Statecharts and Graphical Modeling Framework has been developed and implemented for specifying behaviors of atomic models. One or more Statecharts can be developed for any atomic model and stored in a relational database. The behavioral modeling complements atomic and coupled DEVS structural modeling where families of models with semi-automated code generated for the DEVS-Suite simulator are systematically stored and retrieve. The behavioral modeling enriches visual development of models that have DEVS-compliant specifications with modular, component-based visual representations. The visual representation of hierarchal coupled models are automatically generated or restored. An example model is employed to show the degree of details supported both in visual representation and database representation. Current and future works are briefly described.

Author Keywords
CoSMoS; EMF; GMF; Parallel DEVS; Statecharts; Visual and Persistence Modeling.

ACM Classification Keywords
H.2.4 [Systems]: Relational databases; I.6.1 [Simulation Theory]: Model classification; I.6.4 [Model Validation and Analysis]; I.6.5 [Model Development]: Modeling methodologies; I.6.7 [Simulation Support Systems]: Environments

INTRODUCTION
The Component-based System Modeling and Simulation (CoSMoS) framework was conceived to support a unified logical, visual, and persistent model specification with support for automated simulation code generation for component-based and cellular automata models [1-4]. This modeling engine is integrated with the DEVS-Suite simulator [5], which supports visual experimentation configuration and run-time data collection and observation.

TMS/DEVS 2015, April 12 - 15, 2015, Alexandria, VA, USA © 2015 Society for Modeling & Simulation International (SCS)

The CoSMoS tool enables a simulation-based system design process with support for structural model verification and using simulation validation. CoSMoS stores all models in relational databases.

In DEVS, any component that contains other components is called a coupled model; non-container components are called atomic models. The DEVS formalism [6] includes the means to build hierarchical, modular models from components. A model's behavior is derived from the behavior of atomic models. Behavior of an atomic model is defined by autonomous or input-driven state-based transition functions, which are known as internal and external transition functions.

Beyond basic atomic DEVS, visual behavior modeling can aid in domain-specific applications and particularly those that have complex state-based dynamics. Furthermore, other modeling methods such as standardized Base Object Model component [7] may be enriched with the visual behavior modeling capabilities described in this paper. This is helpful toward making complex model specifications more accessible to domain experts. Visual behavior modeling is also promising for meta-modeling methods such as EMF-DEVS [8].

ATOMIC MODEL BEHAVIOR MODELING
There exists different ways to specify the behavior of an atomic model. Behavior may be defined through mathematics, programming code, or visual notation. Each has its own benefit; the visual notation is more attractive, particularly for domain experts who may not prefer the other approaches. In CoSMoS, the modelers can visually develop the structure of a model, and then automatically translate it to source code for the DEVS-Suite simulator. Behaviors of atomic models must be manually added to the partially generated source code as in any other existing general-purpose visual modeling approach/tool. DEVS-Suite can execute these simulation models.

The UML Statecharts is a standard, powerful and effective visual notation for specifying a system's behavior [9]. At the heart of the Statecharts is the notion of discrete states and the transitions between any two states. Using parallel DEVS is superior to UML Statecharts since the concept of time is explicitly accounted for. On the other hand, unlike DEVS, Statecharts has a rich visual notation. However, neither DEVS nor Statecharts is concerned with persistence modeling (i.e., storing and accessing models).

Given these observations, our research considers three topics. First is supporting visual notation for specifying behavior, second is storing models in a relational database, and third is implementing and integrating these into CoSMoS.

For the first topic, time and ports are required in DEVS, but the handling of time, especially as required for simulation, has shortcomings in UML Statecharts [9, 10]. Handling concurrent events as defined in the confluent function is not accounted for in UML. Simultaneous events are supported in UML2.0, but not in DEVS. Ports are directly handled in the UML2.0 component model.

For the second topic, in software tools such as Rational Rose [11], specifications including Statecharts are stored in flat files, but in CoSMoS all models including their relationships are stored in a relational database [12]. Therefore, new schemas need to be developed for storing visual specifications. Persistence modeling is important for model management, collaborative model development, structural model complexity metrics, and code generation for different target simulation engines[1].

For the third topic, CoSMoS needs to be extended with a new editor for visual behavior modeling. The editor has to support storing and retrieving both structural and behavior models uniformly.

BACKGROUND

Eclipse is an integrated development environment (IDE) that contains a base workspace and an extensible plug-in foundation for customizing and extending it. GMF (Graphical Modeling Framework) is a modeling framework that supports creating a specific graphical editor for logical models. It uses a meta-model following the MVC (Model-View-Controller) architecture. It is based on the EMF (Eclipse Modeling Framework) and GEF (Graphical Editing Framework). GEF is used in tools such as CD++ Builder [13]. EMF handles the model's definition and GEF handles the model's view and controller [14].

GMF provides basic entities for creating graphs consisting of nodes, links, and labels. It can be used to define special types of graphs, such as Statecharts. GMF includes wizards for generating intermediary graphical tools and mapping definitions based on an initial meta-model (EMF Ecore), as well as the final runtime code [14].

Eclipse Modeling Framework

Eclipse EMF can be used to model platform independent software applications from which platform specific code can be generated. EMF distinguishes between meta-models and concrete models. A meta-model describes the structure of the concrete model. Therefore, a concrete model conforms to its meta-model. EMF provides a plug-able framework to store the model information; the default uses XMI (XML Metadata Interchange) to persist (i.e., store) the model definition. EMF allows creating the meta-model using other languages including Java annotations, UML, and XML Schema.

EMF itself is defined using the concept of meta-modeling. It has two meta-models; the Ecore and the Genmodel meta-models. The Ecore contains the information about the defined classes for a domain. The Genmodel contains additional information for the code generation, e.g., the path and file information. The Genmodel also contains the control parameter for code generation. EMF makes domain model explicit, which helps to provide clear visibility of the model. EMF also provides change notification functionality to the model in case the model changes. EMF will generate interfaces and factory classes for creating objects; therefore it helps to keep the application clean from the individual implementation classes. Another advantage is the ability to regenerate the Java code from the model at any point in time.

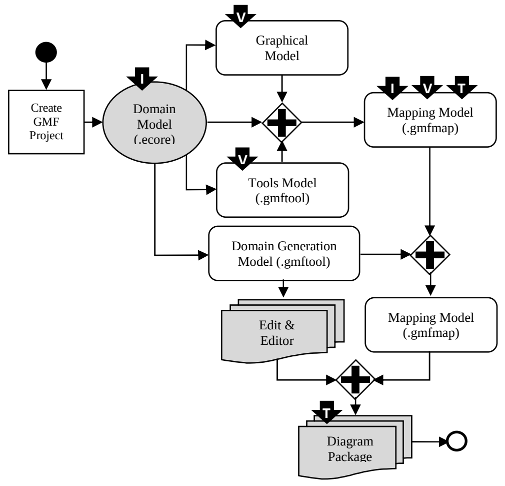


Figure 1. GMF Workflow Process.

Graphical Modeling Framework

The GMF provides a workflow for generating graphical editors (see Figure 1). The starting point is to define a domain model using the Ecore Domain Generation Model, which generates the implementation code. The graphical model defines the symbols used for representing different object classes visually (as nodes), as well as labels for displaying and editing attribute values. Classes and references may also be mapped to *connections* (links between nodes), and some references may be defined as *containment* for decomposition of a node into sub-nodes in *compartments*. A wizard helps generate a basic graphical model from the domain model, though custom figures and decomposition must be added manually.

The tools model defines the services for creating new nodes and connectors in the model editor. A wizard helps selecting which of the object classes in the domain model

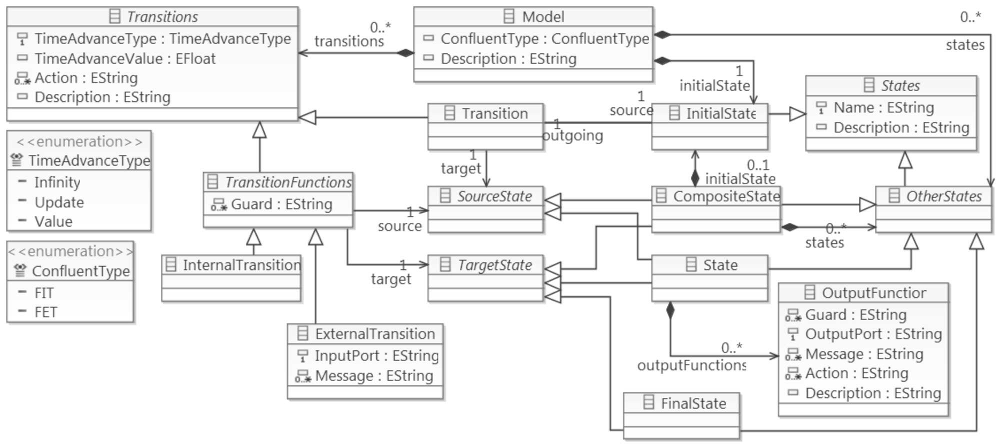


Figure 2. An Ecore DEVS meta-model using Statecharts.

will be available. The mapping model connects the domain, tools, and graphical models, defining which node and connector symbols are used for representing which domain model class and which tools create new elements. Indeed, this part makes the relation between graphical elements, tool elements and their classes. By using the GMF wizards, the basic mappings are established automatically, though decomposition rules and links between different diagrams must be added manually.

The Java code for editing diagrams is generated from the mapping model. It uses the code generated from the domain model. The dark arrows embedded with I, V, and T characters in Figure 1 reflect the additional configuration of the model editor that the modeler may input at each stage. The domain model reflect the information content aspect, while the graphical and tools models represent the view aspects of model elements and tools (tasks or services), respectively. I, V, and T are for the information, view and task aspects. Additional tasks or services can only be defined through programming. The tools model identifies services for creating new model elements, but all other kinds of tasks are implicit. The Diagram Generation Model configures GMF to generate diagramming code. The purpose of this model is similar to the Domain Generation Model.

STATECHARTS EDITOR

EMF Ecore model is the basic model to define the structure of a desired visual editor in GMF. Figure 2 depicts the Ecore model (Class Diagram) for the atomic DEVS model. There is ConfluentType attribute in *Model* class to define confluent function. States in DEVS are defined using four classes; *InitialState*, *State*, *CompositeState* and *FinalState*. *InternalTransition* and *ExternalTransition* classes capture the Internal Transition Function and External Transition Function, respectively. To initialize the model, we add a *Transition* class which just starts from an *InitialState*. The *OutputFunction* class defines the output function of DEVS that just can be added to the *State*. TimeAdvanceType and TimeAdvanceValue attributes in the *Transitions* class define to show a simple time advance function. *States*, *OtherStates*, *SourceState*, *TargetState*, *TransitionFunctions* and *Transitions* are abstract classes.

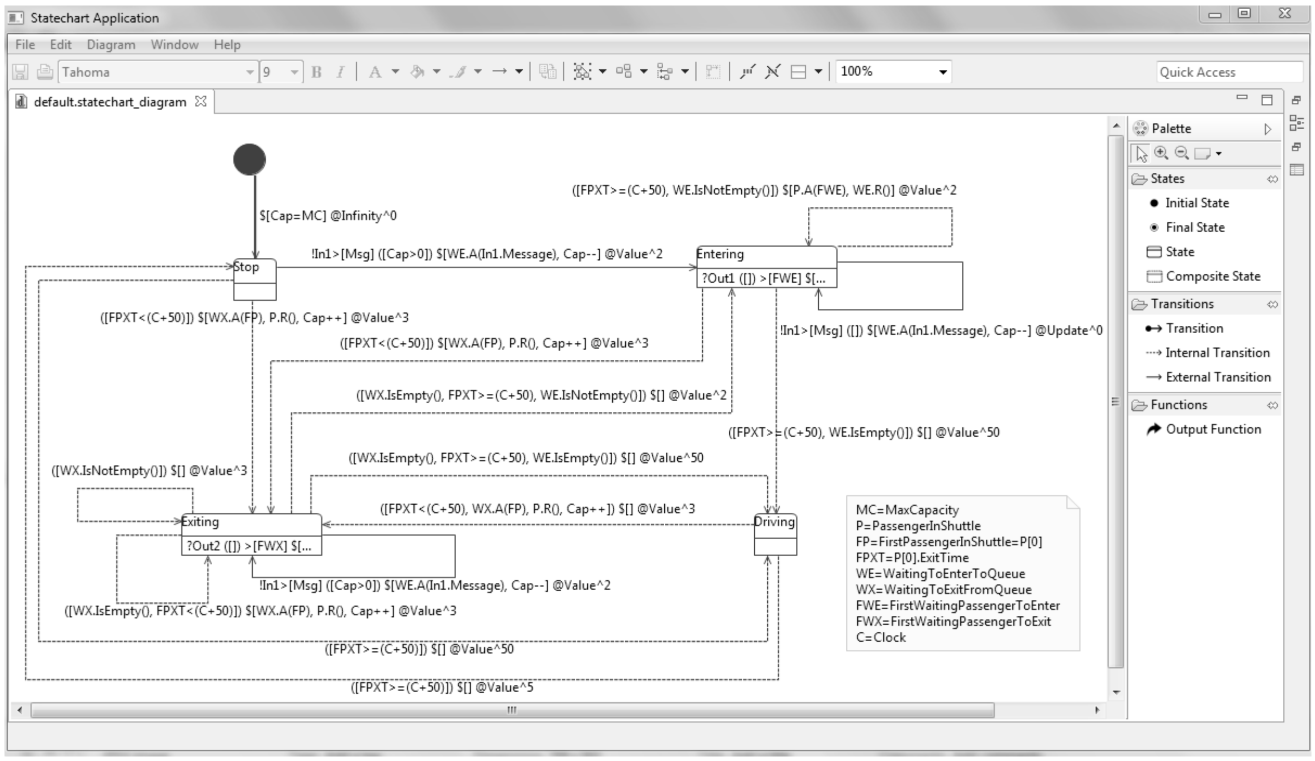
We can define some rules for the DEVS atomic model specification in the Statecharts just described fined. For example, we just can have one initial state in a model and/or any composite state. We observe this rule by adding initialState connections (EReferences connections) from *Model* and *CompositeState* classes to *InitialState* class with Upper Bound property set to one. Initial state is just the source state of *Transition*; also target of *Transition* just can be form *SourceState* (State or Composite State).

Final state cannot be the source state of any transition. This rule is observed by adding two abstract classes (*SourceState* and *TargetState*) and define that *FinalState* just inherit from *TargetState*. Any internal or external transition has source state (State or Composite State) and target state (State, Composite State or Final State). Any composite state is like a new model which can have its initial state, states, composite states, etc.

We have developed an editor to visually specify the behavior of atomic DEVS model by applying the steps of the GMF workflow. Figure 3 is a snapshot of the generated Statecharts Editor (SE) that is an RCP Application. The editor is divided into main canvas (left side / drag and drop environment), toolbox (right side), and property window. Models are created in the main canvas. Menu bar and Tool bar are located in the top, and there are some tools for arranging elements in the main canvas. Position and size of all windows can change manually.

STATECHART EDITOR ELEMENTS PROPERTIES

The Statecharts Editor's meta-data (Ecore Model) and generated editor has specific entities to visually model the behavior of atomic models, thus the behavior of the system. Although certain parts of visual behavior model development are supported by the Ecore in Figure 2, some other parts still have to be completed in code. For example, guards are implemented by simple string properties; thus, if

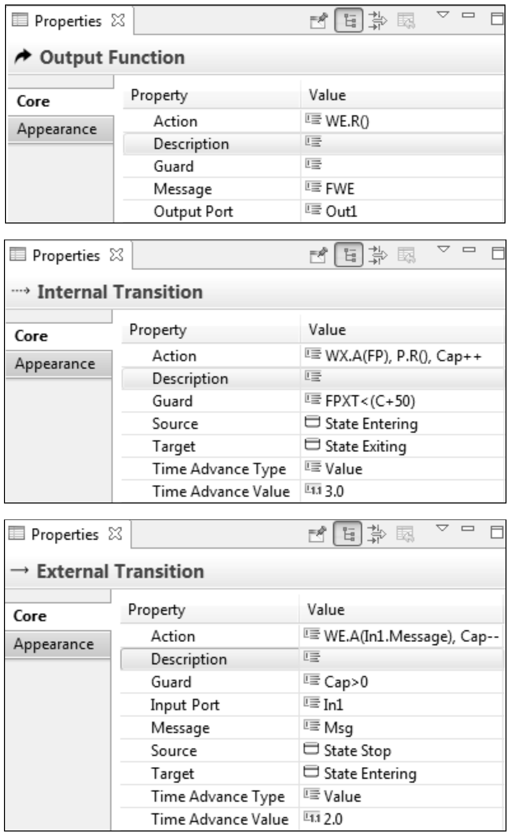

Statechart Application
File Edit Diagram Window Help
Tahoma
9
100%
Quick Access
default.statechart_diagram
$[Cap=MC] @Infinity^0
([FPXT>=(C+50), WE.IsNotEmpty()]) $[P.A(FWE), WE.R()] @Value^2
!In1>[Msg] ([Cap>0]) $[WE.A(In1.Message), Cap--] @Value^2
Stop
Entering
?Out1 ([]) >[FWE] $[...
([FPXT<(C+50)]) $[WX.A(FP), P.R(), Cap++] @Value^3
!In1>[Msg] ([]) $[WE.A(In1.Message), Cap--] @Update^0
([FPXT<(C+50)]) $[WX.A(FP), P.R(), Cap++] @Value^3
([WX.IsEmpty(), FPXT>=(C+50), WE.IsNotEmpty()]) $[] @Value^2
([FPXT>=(C+50), WE.IsEmpty()]) $[] @Value^50
([WX.IsEmpty(), FPXT>=(C+50), WE.IsEmpty()]) $[] @Value^50
([WX.IsNotEmpty()]) $[] @Value^3
([FPXT<(C+50), WX.A(FP), P.R(), Cap++]) $[] @Value^3
Exiting
?Out2 ([]) >[FWX] $[...
Driving
!In1>[Msg] ([Cap>0]) $[WE.A(In1.Message), Cap--] @Value^2
([WX.IsEmpty(), FPXT<(C+50)]) $[WX.A(FP), P.R(), Cap++] @Value^3
([FPXT>=(C+50)]) $[] @Value^50
([FPXT>=(C+50)]) $[] @Value^5
Palette
States
Initial State
Final State
State
Composite State
Transitions
Transition
Internal Transition
External Transition
Functions
Output Function

MC=MaxCapacity
P=PassengerInShuttle
FP=FirstPassengerInShuttle=P[0]
FPXT=P[0].ExitTime
WE=WaitingToEnterToQueue
WX=WaitingToExitFromQueue
FWE=FirstWaitingPassengerToEnter
FWX=FirstWaitingPassengerToExit
C=Clock

Figure 3. Statecharts Editor.

there is a need to define a function, it has to be manually added to the generated code. Each simple and complex entity has a unique visual icon with one or more properties. An example of a simple entity is State defined to have a name with data type string. The Initial State, Final State, Composite State and State just have Name property. It has to be filled before saving a model. Actions, Guards and Messages are arrays of String. Action uses to set states as assignments, Guard uses to check conditions, and Message uses to define output values, to send by output function.

In Transition, the SourceState can be an initial state, and TargetState can be a state or composite state. In the Internal and External Transition, the Source property can be a state or composite state; and the Target property can be a state, composite state, or final state. The External transition activates by receiving message on its Input Port. The output port with message/messages is set by Output Port property in the output function. Specifications for the internal, external, and output functions are shown in Figure 4. The Time Advance Type can be set to Infinity, Update or Value. Time Advance Value sets to a value ranging from 0 to less than infinity when Time Advance Type sets to Value. An example of specifying the time advance with a finite value is shown in the External Transition in Figure 4. For the output function, Message property is defined to be FWE and the port to be out1. An action can be a simple assignment or complex (see the internal and external transition functions). The confluent function is defined to be either FIT (First Internal Transition) or FET (First External Transition).



Output Function

Property	Value
Action	WE.R()
Description	
Guard	
Message	FWE
Output Port	Out1

Internal Transition

Property	Value
Action	WX.A(FP), P.R(), Cap++
Description	
Guard	FPXT<(C+50)
Source	State Entering
Target	State Exiting
Time Advance Type	Value
Time Advance Value	3.0

External Transition

Property	Value
Action	WE.A(In1.Message), Cap--
Description	
Guard	Cap>0
Input Port	In1
Message	Msg
Source	State Stop
Target	State Entering
Time Advance Type	Value
Time Advance Value	2.0

Figure 4. Sample specifications for the internal, external and output functions.

ATOMIC MODEL STORAGE

The data for every atomic model developed in the Statecharts Editor is stored in two flat files. The *Domain file* has the model's data, such as the atomic model's attributes and functions. The *Diagram file* has the model's graphical attributes, such as position, font, and color for every element of the model.

In contrast, CoSMoS stores every model in a database without any data about their graphical representations. The CoSMoS client generates visual representations of the models using a set of rules. For CoSMoS, it is useful to store the behavioral model (i.e., the Domain file) with its corresponding structural model. One option is to use existing plug-ins, such as Teneo, Hibernate, and CDO [15]. Another option is to modify the Diagram Generation Model package in GMF to store the Statecharts model in the CoSMoS database. Table 1 lists some of the basic pros and cons that pertain to this work.

	Plug-ins Architecture	Classical
Pros	Automatic database (re-) generation	Full control in designing and implementing databases
	Built-in multi-client support with minimal manual programming	Flexibility in using databases with existing client applications
Cons	Dependency on complex plug-ins and Eclipse Framework	Manual development

Table 1. Model Persistence Approaches in Databases.

Based on our requirements, we have chosen the classical approach for adding the Statecharts model to the existing CoSMoS database. In particular, automatic database generation using plug-ins does not offer a tangible benefit since the SE is not subject to user-specific changes.

There are two approaches to achieve the storing data mechanism. The first is to read and modify all the classes that are automatically generated by GMF and the second is to implement the needed changes in the CoSMoS without making any changes to the SE. We decided to choose the second approach due to the complexity of the GMF's plug-ins.

As mentioned above, every model in the SE has the Domain and Diagram files. The names of the domain and diagram models are the same as the statechart's name which defined by modeler. There is no need to store the properties in the Diagram file in separated tables.

Figure 5 shows the schema for storing Statecharts models. The CoSMoS database is extended with this schema. The primary key for these tables are `AMname` (Atomic Model name) and `SCname` (Statecharts name). When a modeler saves a model, its domain file is parsed and stored in the database tables. The database is used to recreate the Domain and Diagram files that are needed in the SE. The SE works as a standalone application; it uses flat files whose content are identical to the data contained in the CoSMoS database.

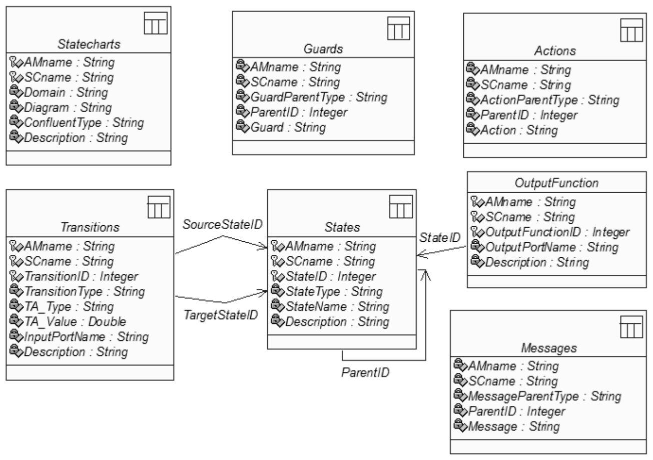


Figure 5. Database schema to storing DEVS atomic model logical and visual specifications.

When a modeler wants to save one or more Statecharts models in the database, CoSMoS will parse the domain file to extract the relevant data that is to be stored in the database. Figure 6 illustrates the sequence diagram to extract and save data, for states and transitions. It is the same for other model's elements such as Guards, Actions, Messages, and Output Functions.

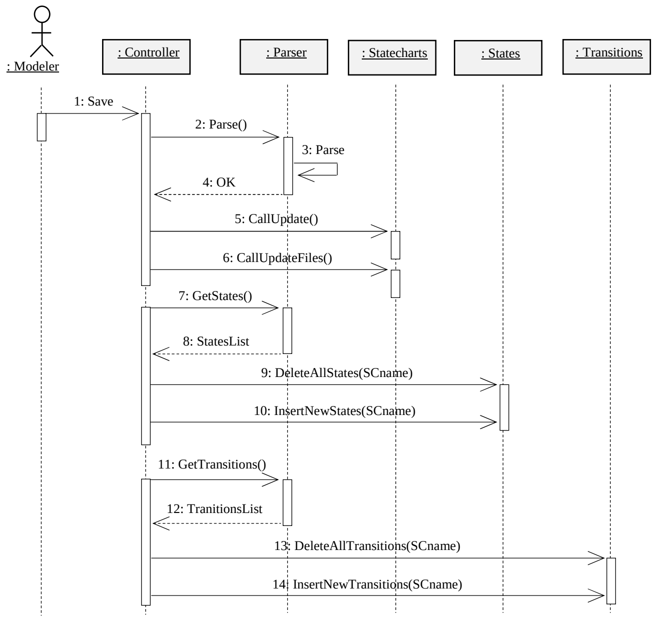


Figure 6. Sequence diagram for storing behavioral atomic models in database.

When a modeler wants to save a model, the `Controller` object calls `Parse()` method of the `Parser` object. It parses the domain file, creates a list of elements and returns OK. The `Update()` and `UpdateFiles()`methods copy the whole content of the domain and diagram files into the domain and diagram fields of the Statecharts table. Then, all instances of an element are returned with their properties on a list (e.g., States). Next, the current data, which is for a specific atomic model, is deleted and replaced with new data. Similar steps are executed for all the elements of a Statecharts model.

Figure 7 illustrates the sequence diagram for opening an atomic model from the database. When a modeler wants to open an atomic model, one or more Statecharts model will be opened on individual tabs in the SE (see Figure 3). At first, all the Statecharts of an atomic model are read from the Statecharts table in database. Then, the Domain and the Diagram flat files are created in their pre-specified location. Finally, new tabs will be created in the main canvas of the SE and automatically filled with the data retrieve from CoSMoS database.

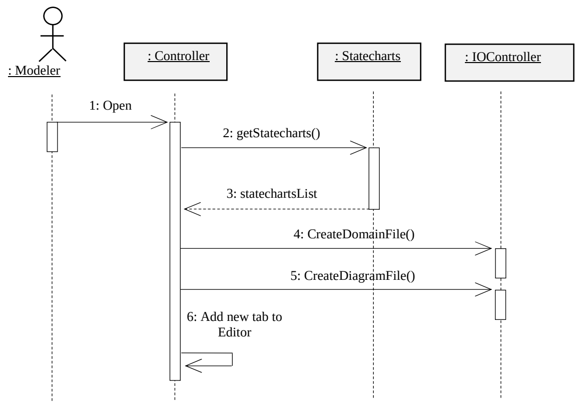


Figure 7. Sequence diagram to open a model.

Communications between the SE and the CoSMoS with the database are different. In CoSMoS all validations are checked against data which are stored in the database. In the SE validations are enforced by the editor, by the database, and by the domain flat file.

INTEGRATING COSMOS AND STATECHARTS EDITOR

Until now, we have two visual editors for atomic DEVS models (one for developing structural models and another for developing behavioral models). These modeling environments work independently, but use a shared database. It is important for these two environments to be integrated and have a workflow with the ability to iteratively and incrementally develop models. The workflow must ensure both environments use the database consistently and in particular guarantee that the database and the SE flat files remain consistent (i.e., the database and flat files are not changed simultaneously).

The first approach was to embed the SE within CoSMoS. However, since the SE is an RCP Application, it could not be included in CoSMoS, a Java Application. This is due to the SE launching with eclipse configurations which cannot be supported in Java applications. Also, we could not find a way to convert the RCP application to a Java application. As an alternate solution (second approach), we devised a "calling mechanism" where CoSMoS is embedded in the SE.

Figure 8 details the workflow between the CoSMoS Java application (referred to as CoSMoS$_A$) and the SE. The integration required converting the Statecharts editor RCP Application to its executable counterpart. It is called CoSMoS$_R$. It supports both structural and behavioral atomic DEVS modeling while retaining the complete range of modeling already supported in CoSMoS$_A$. In this setting, an instance of the CoSMoS$_A$ is launched as a thread in CoSMoS$_R$ which is the master thread. The part of Figure 8 enclosed in the dotted rectangle is for CoSMoS$_A$. The remaining steps and flows are for the master thread. The atomic model for which Statecharts models are to be developed is selected in CoSMoS$_A$. Then, the Statecharts can be developed (see the "Wake up RCP's thread" step in Figure 8).

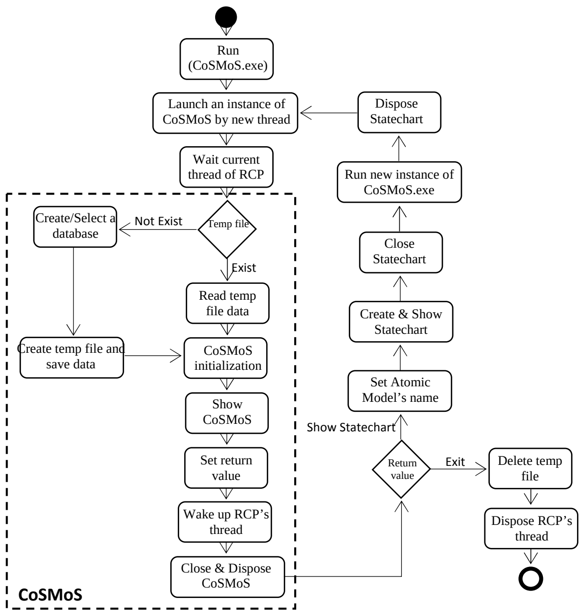


Figure 8. Model Development Workflow Process.

The CoSMoS$_R$ thread looks for a temporary file (CurrentDB.xml) along with the CoSMoS$_R$. If it does not exist, the modeler must either create a database or select an existing one. Information about the selected database and the path to its flat files will be stored in the `CurrentDB.xml` temporary file. Then CoSMoS$_A$ is initialized and launched. If a temporary file exists, information will be read from it and automatically initialized. The CoSMoS$_A$ visual editor is displayed and the modeler can modify the structure of the model. The modeler has two options: launch the SE or terminate CoSMoS$_R$. In the former scenario, the CoSMoS$_A$ thread sets appropriate results for waking up the CoSMoS$_R$ thread, which launches the SE. When CoSMoS$_A$ is to exit, then the `CurrentDB.xml` will be deleted and the master thread will be disposed. Otherwise, the SE is launched with the name of atomic model. If there are any behavioral atomic models for the named atomic model, they are loaded in the SE and the modeler can change and/or add new Statecharts models.

As shown in Figure 9, any behavior atomic model has a defined Template Model (TM). When an atomic model is selected from the TM tree, new Statecharts behavior can be developed for it or, alternatively, existing models can be modified or deleted. Every Statecharts has a unique name. The modeler can select and delete any model from a list of Statecharts. The database and flat files are updated and removed accordingly.

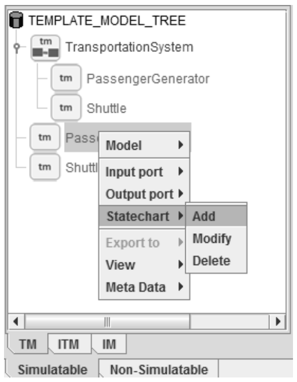


Figure 9. Statecharts modeling in CoSMoS.

RELATED WORKS

Visual behavior specification for atomic DEVS model behavior has been of interest for many years. States and transitions of atomic models are commonly used to visually represent DEVS model behaviors of atomic models with tools supporting varying levels of specifications. A common approach is to use State Machines. The behavior of a generic classic atomic model is reduced to the syntax and semantics afforded by the State Machines. Another approach for behavior modeling is Statecharts [16]. There are different ways of representing (mapping) the behavior of atomic models to Statecharts (e.g., [10, 17]). Another approach is developed for CoSMoS (see Figure 3). Atomic model behavior may also be defined using custom notations (e.g., [18]). Recent approaches and tools that use or adapt basic State Machines include CD++ Builder [13], LSIS DME DEVS [19], DEVS Graph [20], and MS4Me [21]. Other simulation tools such as Simulink [22] and Ptolemy II [23] use variants of State Machines.

Amongst the above approaches, Statecharts is more attractive since it is a visual language and more expressive as compared with State Machines and its variants such as Stateflow [22]. Statecharts is widely adopted and is included in the UML standard. The SE has a well-defined UI partitioning which simplifies model development, for example, through the palettes and property panels (as opposed, for example, to use of dialogue boxes).

The visual behavior modeling supported in CoSMoS tool has similarities and differences with those mentioned above. Basic similarities with the above tools are defining states and transitions at the Statecharts (and therefore State Machines) abstraction level. Key differences are the specification of input and output ports. The external and internal transitions for DEVS atomic model are directly specified in terms of these ports. In the context of CoSMoS as an integrated modeling and simulation framework, families of models in a disciplined setting, multiple behavioral models for a model may be constructed. This capability leads to specializing an atomic model in terms of not only its structure, but also its behavior.

In comparison to other approaches, including the tools mentioned above, models in CoSMoS are stored in relational database. Disciplined model persistence is crucial when modelers are faced with creating alternative models [24]. Traditional methods of storing models in flat files can become unnecessarily complex when more than a few model components are to be developed. Synthesizing a family of models visually is attractive especially as an atomic may have different behaviors in different coupled models. From a broader perspective, CoSMoS is compared with MS4Me as an exemplar (commercial) DEVS and Ptolemy II as an exemplar (open source) component-based tool. Structural and behavioral DEVS models are stored in relational database, whereas in MS4Me and Ptolemy II structural and behavioral models are stored in flat files. Among these tools, CoSMoS and MS4Me frameworks can support similar, yet distinct, approaches for creating families of models.

CONCLUSIONS

The work presented in this paper introduces a rich visual behavior modeling capability to the Component-based System Modeling and Simulation (CoSMoS) environment. As described in the earlier sections, specifying the states, transitions, and timing for atomic models as Statecharts is developed using the Graphical Modeling Framework (GMF). The resulting Statecharts Editor offers a rich environment to develop non-trivial behavior models. DEVS behavior model syntax is readily captured and supported in GMF. Furthermore, these behavioral atomic models, alongside their structural counterparts, are stored in the CoSMoS relational database. Consequently, both structural and behavioral DEVS modeling is supported. For future work, it is useful to support code generation behavioral models for the DEVS-Suite simulator. This requires extending structural simulation code generation with the external, internal, confluent, time advance, and output functions. A benefit of this generated code is that basic syntactic specification of atomic models can be automatically verified.

REFERENCES

1. H. S. Sarjoughian and V. Elamvazhuthi, 2009, "CoSMoS: a Visual Environment for Component-based Modeling, Experimental Design, and Simulation," *in Proceedings of the 2nd International Conference on Simulation Tools and Techniques.*

2. H. S. Sarjoughian, V. Elamvazhuthi, and S. Sarkar, 2002-2009, "CoSMoS 2.1.0 Help Document/Guide,"
3. Arizona Center for Integrative Modeling and Simulation, 2014, http://www.acims.arizona.edu.
4. CoSMoSim Verions 3.0, 2009, http://cosmosim.sourceforge.net.
5. DEVS-Suite Simulator Version 3.0, 2015, http://devs-suitesim.sourceforge.net.
6. B. P. Zeigler, H. Praehofer, and T. G. Kim, 2000, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Dystems*, 2nd Ed, San Diego: Academic Press.
7. B. O. M. SISO, 2006, "Base Object Model (BOM) Template Specification. *Simulation Interoperability Standards Organization*," SISO-STD-003-2006.
8. H. S. Sarjoughian and A. M. Markid, 2012, "EMF-DEVS modeling," *in Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*.
9. OMG Unified Modeling Language, 2004, http://www.omg.org/docs/formal/05-07-04.pdf.
10. J. Mooney and H. S. Sarjoughian, 2009, "A Framework for Executable UML Models," *in Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*.
11. IBM - Software - Rational Rose family, 2014, http://www.03.ibm.com/software/products/en/ratirosefami.
12. T. S. Fu, 2002, "Hierarchical Modeling of Large-scale Systems Using Relational Databases," The University of Arizona.
13. M. Bonaventura, G. Wainer, and R. Castro, 2010, "Advanced IDE for Modeling and Simulation of Discrete Event Systems," *in Proceedings of the 2010 Spring Simulation Multiconference*.
14. R. C. Gronback, 2009, *Eclipse Modeling Project: a Domain-specific Language Toolkit*. Upper Saddle River, NJ: Addison-Wesley.
15. CDO/Hibernate Store - Eclipsepedia, 2014, https://wiki.eclipse.org/CDO/Hibernate_Store
16. D. Harel, 1987, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, 231-274.
17. S. Borland, 2003, "Transforming Statechart Models to DEVS," McGill University.
18. U. B. Ighoroje, O. Maïga, and M. K. Traoré, 2012, "The DEVS-driven Modeling Language: Syntax and Semantics Definition by Meta-modeling and Graph Transformation," *in Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*.
19. M. Hamri and G. Zacharewicz, 2012, "Automatic Generation of Object-oriented Code from DEVS graphical specifications," *in Proceedings of the Winter Simulation Conference*.
20. H. S. Song and T. G. Kim, 2010, "DEVS Diagram Revised: A Structured Approach for DEVS Modeling," *in Proceedings of the European Simulation Conference*.
21. C. Seo, B. P. Zeigler, R. Coop, and D. Kim, 2013, "DEVS Modeling and Simulation Methodology with MS4 Me Software Tool," *in Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*.
22. G. Hamon and J. Rushby, 2004, "An Operational Semantics for Stateflow," *Fundamental Approaches to Software Engineering*, 229-243, Springer Berlin Heidelberg.
23. Ptolemy II Home Page, 2014. http://ptolemy.eecs.berkeley.edu/ptolemyII/.
24. H. S. Sarjoughian, J. J. Nutaro, and G. Joshi, 2011, "Collaborative Component-based System Modeling," *Journal of Simulation*, Vol. 5, No. 2, 77-88.