CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation

Hessam S. Sarjoughian
Arizona Center for Integrative Modeling & Simulation
Computer Science and Engineering Department
Arizona State University, Tempe, AZ 85281-8809
001-480-965-3983
sarjoughian@asu.edu

Vignesh Elamvazhuthi
Content Management and Archiving
EMC Corporation
Pleasanton, CA 94566
001-925-600-5884
elamvazhuthi_vignesh@emc.com

ABSTRACT
An integrated modeling and simulation tool called Component-based System Modeler and Simulator (CoSMoS) is developed. It supports visual development of families of models that have well-defined logical specifications. The logical component-based models persist in relational databases and may be automatically translated into specific target simulation and markup programming languages. The underlying system-theoretic modeling framework of CoSMoS lends itself for the well-known discrete-time, continuous, and discrete-event modeling approaches. Currently, CoSMoS supports developing parallel DEVS-compliant models which can be executed using the DEVS-Suite simulator. The underlying process lifecycle of the CoSMoS enables systematic transitioning from visual model development and design of experiments to simulation execution and experimentation. Simulation data can be used for run-time animation and viewing of time-based trajectories or exported for post processing. This tool helps to simplify simulation-based system design, verification, and validation. The core capabilities of the CoSMoS are exemplified with a conceptual model of an anti-virus network software system.

Categories and Subject Descriptors
I.6.1 [**Simulation and Modeling**]: Types of Simulation – *animation, visual*; I.6.5 [**Simulation and Modeling**]: Model Development – *modeling methodologies*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems – *environments*

General Terms
Design, Experimentation, Measurement, Theory, Verification.

Keywords
CoSMoS, M&S lifecycle, DEVS-Suite, visual modeling

SIMUTools 2009, March 2–6, 2009, Rome, Italy.

1. INTRODUCTION
Complex systems are described using a set of model abstractions and relationships. For example, the Unified Modeling Language (UML) [9] and Discrete Event System Specification (DEVS) [14] are primarily used for software modeling and simulation modeling, respectively. The abstractions and relationships offered by these allow modeling both structures and behaviors of dynamical systems. In contrast, other languages such as XML Schema and System Entity Structure (SES) [13] are mainly targeted for modeling structures of systems. XML Schema [3] can be used to describe arbitrary data structures among simple and complex elements. Families of models may also be specified using an extensive set of elements with pre-defined and user-defined rules. Similarly, SES supports data modeling, but has a fixed set of rules that constrain how entities' (objects without behavior) abstractions may be organized. Among these approaches, UML provides a unified logical and visual modeling framework. Numerous other efforts and tools have been proposed and undertaken, but many lack sound underlying principles that can empower users to visually develop logical models and automatically translate into simulation code and simulated.

It is desirable to describe systems using logical and visual model types that can also lend themselves for simulation. Logical models can be mathematical formulations of a system's structure and behavior and are important since they have precise syntax and semantics. Visual models, on the other hand, are desirable since they help simplify modeling, especially for domain experts who find formal specifications or programming difficult and not intuitive. Furthermore, it is desirable to represent models in standard languages that are well suited for databases. Models can be used as persistent repositories and therefore be systematically reused for multiple purposes and evolved over time and space during the lifecycle of simulation models.

Given the unique capabilities and advantages afforded by logical, visual, and persistence models, it is advantageous to have a modeling framework that supports collective use of these different model types in a consistent manner. That is to say, all visual model development activities must be sanctioned by the logical models and all models that are stored in a database must be consistent with their logical specifications and thus their

visual representations. The Component-based System Modeling (CoSMo) framework satisfies the above requirement.

A key disadvantage for visual modeling is limited viewing space. Techniques such as hierarchical modeling with zoom-in and zoom-out capabilities can significantly reduce the viewing space limitation. However, a user who is interested in developing models and simulating them needs to design experiments. For observing model components' input and output data, it is useful to support visual selection of the components and their individual input and output variables. To support such a capability, the concept of visual design of experiments is proposed and introduced into the CoSMo framework. The resulting Component-based System Modeling and Simulation (CoSMoS) framework is used to develop the Component-based System Modeler and Simulator environment which integrates the CoSMo modeler and DEVS-Suite simulator together. The design of the CoSMoS has a lifecycle process in which a user starts with visual model development and design of experiments and executes simulation models that are partially generated and manually completed by user.

In the remainder of this paper, we briefly present the basic modeling approach of CoSMo, the complementary viewing of model executions supported by the DEVS-Suite simulator, and an example that helps to illustrate integrated model development and simulation in CoSMoS. Next, we describe the basic design of CoSMoS that ensures visual model configuration for simulation experimentations, and automated data collection and viewing. In the following two sections, we present the process lifecycle and related work. We conclude with a summary of the paper and future work.

2. BACKGROUND

In this section, we provide a brief account of CoSMo and DEVS-Suite with emphasis on aspects that are important in integrating into the CoSMoS environment. We also describe an example that can illustrate the kind of end-to-end modeling and simulation that is supported in CoSMoS.

2.1 CoSMo

Component-based System Modeler (CoSMo) is a tool [1] for developing a family of models of a system [2][3] [5][10][11]. It has a unified concept for specifying general-purpose logical, visual, and persistent primitive and composite models (see Figure 1). Complex hierarchical models may be developed by composing modular components using their input and output ports. CoSMo currently supports DEVS and XML models and generates partial and complete source code for DEVS-Suite [1][6].

The logical model specification is governed by a set of axioms that ensure consistency among a family of alternative hierarchical model specifications [5][10]. A system is modeled in terms of three model types: Template Models (TM), Instance Template Models (ITM) and Instance Models (IM). All primitive and composite logical, visual, and persistent models are defined in terms of these model types. The primitive and composite models can collectively represent alternative specifications of one or more systems. Every Instance Template Model is defined only when it has a Template Model and every Instance Model is defined only when it has an Instance Template Model.

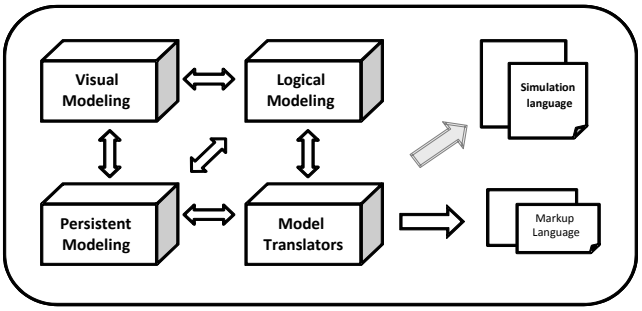


Figure 1. Architecture Concept for CoSMo

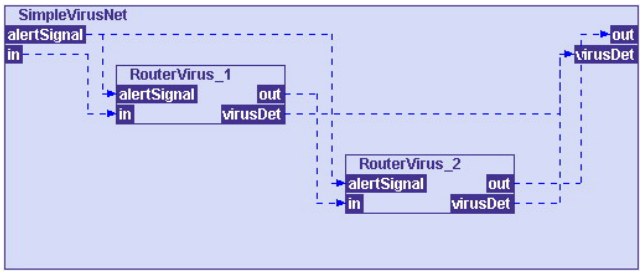
2.1.1 Logical Models

The Template Model is defined to include primitive and composite models with hierarchy of length two. The composition and specialization relationships may be used together under some well-defined constraints (e.g., absence of self-composition as defined in the system-theory and self-feedback as defined in DEVS formalism) to specify strict hierarchical tree structures of one or more models. To avoid possible confusion, composition refers to composite/primitive (or whole/part) relationships (i.e., a composite component can contain one or more primitive components). The specialization refers to parent/child relationship where a primitive or a composite component can be specialized to a primitive or a composite component, respectively. A model can be a specializee component in which case it can have one or more specialized components. There is a specialization relationship between any specialized component and its specializee component. With the Template Models, separate models can be specified for systems that may or may not be related to one another. Limited behavioral modeling (specifying input and output variables) is supported and structural complexity metrics (e.g., number of components for any composite model) can be readily obtained.

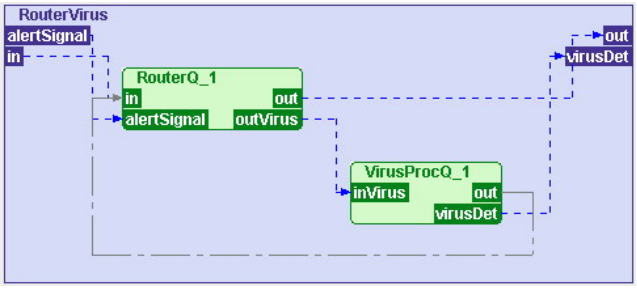
The Instance Template Model, which extends the Template Model, is defined to have hierarchies with lengths equal or greater than two. Multiple Template Models can be combined together to form different models of a system. The Instance Template Model represents an instance of the Template Model since Template models may be combined using the composition and specialization hierarchies to specify alternative structures of a system. Although any two Instance Template Models are distinct, they may share one or more Template Models. The use of the term instance is not in the sense of Object Theory where all instances of a class have an identical specification. Two Instance Template Models differ in terms of their specializations and compositions. The Instance Model is defined to represent structures that do not have any specialization relationships. An instance model is instantiated by removing all specialization relationships (selecting specialized components from specializee components) that may be contained in an Instance Template Model.

Since there can be many alternative models for a system, it is important to keep them consistent with one another. A concept that is commonly used in specifying hierarchical models of systems is uniformity. A model for a part of a system (i.e., primitive or composite model) that is used multiple times in the system's model hierarchy must have a unique specification (i.e., structure and behavior). When the structure of a model is restricted to be a tree instead of a graph, uniformity implies that for any sub-structure with a unique specification and name, all of its occurrences are identical. A consequence of enforcing this property is that changes made to the sub-structure are uniformly applied to the complete tree structure. Another property called non-self-reference states that a model cannot contain itself either directly through iterative composite/primitive relationships. Based on the above relationships (i.e., composition and specialization) and properties (uniformity and non-self-reference), a finite set of unique Instance Models can be generated given the Template Models and Instance Template Models.

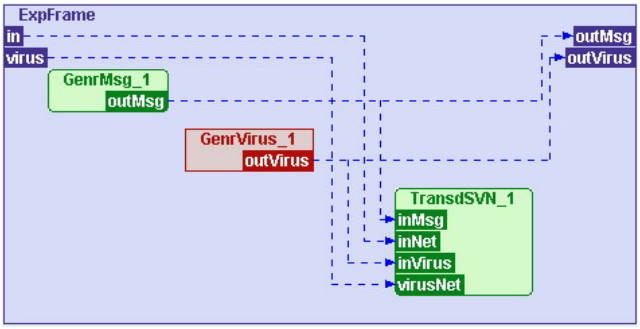
Instance Models are concrete since they cannot have any specialization relationships. The transformation relationship (between the Instance Template Model and the Instance Model) enables defining one or many structures that are defined by removing all occurrences of specialization relationships that may be contained in the Instance Template Model. Thus, a family of unique instance models can be generated from the Instance Template Models.



(a) Composite virus network model



(b) Primitive router and virus processor models



(c) Composite experimental frame model with specializee generator virus model

Figure 2. Visual Template Model development in CoSMo

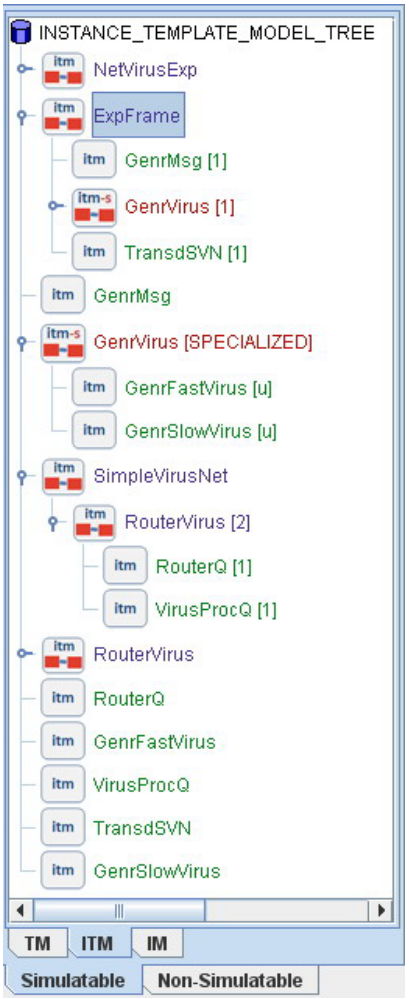


Figure 3. Instance Template Model for virus network model

2.1.2 Persistent and Visual Models

The CoSMo environment supports storing models in relational databases. Model creation, storage, access and manipulation require management of a large number of models and determining their structural complexity metrics [9]. The visual modeling supports developing and manipulating composite models that are synthesized from primitive and composite models. Specification of primitive and composite models are based on block models that can be combined using input and output ports and links that connect them together using specific modeling languages such as DEVS.

2.1.3 Model Translator

The translator for CoSMo supports translating the logical models that are stored in any of its databases to code for target simulation and markup languages. In general, since it is impractical to visually model arbitrary dynamical models, only partial source can be automatically generated. Logical primitive and composite models can be automatically translated to atomic and coupled models for DEVS-Suite simulator. The translator generates partial source code for DEVS atomic models from primitive Instance Models and complete source code for DEVS coupled models from composite models. It is for this reason the shaded arrow is used between the Model Translators to Simulation Languages (see Figure 1). Translators have also been developed for generating DTD and XML schema code.

2.2 DEVS-Suite

DEVS-Suite [6] is the next generation of the DEVSJAVA simulator [1]. It supports simulating models that can be specified using the DEVS formalism [14]. The architecture of the simulator is based on Model Façade View Control (MFVC) and in particular has its animation and viewing of time trajectories separated from the parallel DEVS abstract simulator. Models in DEVS are classified into atomic and coupled models. An atomic model is defined in terms of input, output, state, and time sets with functions that determine next states and outputs given current states and inputs at arbitrary time instances. Together, external, internal, confluent, output, and time advance functions define a component's behavior over time. A coupled model is defined in terms of atomic and/or coupled models. As in an atomic model, a coupled model contains a set of inputs, a set of outputs, a set of component names, a set of components, and a set of coupling relationships among the input and output ports of the composed model components. Atomic and coupled models interact with one another using messages that are exchanged via couplings that connect their input and output ports.

The execution of the models can be viewed as the animation of the input/output messages for coupled models and the state changes of the atomic models. For any atomic or coupled models, its inputs and outputs can be selected via a dialogue box at the beginning of the simulation and time-based trajectories generated during simulation. For atomic models, trajectories can also be generated for pre-defined phase and sigma state variables.

2.3 Anti-Virus Network Software System Example

To illustrate model development and simulation in CoSMoS [1], we have constructed a model called SimpleVirusNet for a simple virus detection software system which is stored in a database file called NetVirs_Net.mdb (see Figure 2(a)). The system is intended to protect a network of computers from virus attacks. A model of this system which is called SimpleVirusNet consists of two coupled models called RouterVirus. The messages arriving at the in port of the SimpleVirusNet are sent to the in port of the first RouterVirus model. The messages arriving at the SimpleVirusNet alertSignal are sent to the alertSignal of both RouterVirus models.

Each RouterVirus has one RouterQ and one VirusProcQ components (see Figure 2(b). The RouterQ acts as a processor. If it receives a message which is not infected by a virus, the message is sent to the out port. The type of messages arriving at the alertSignal port is the same as the messages arriving at the in port. A message arriving at the in port is considered to be suspicious if its ID matches the ID of the message arriving at the port alertSignal. The RouterQ has two queues, q and alertQ, to store the messages and the alert messages respectively.

An experimental frame model called ExpFrame is defined. As shown in Figure 2(c), this model consists of the GenrMsg, GenrVirus, and TransdSVN models. The GenrMsg generates messages for the in port of the SimpleVirusNet and GenrVirus generates messages for the alertSignal port. The TransdSVN computes statistics such as throughput for the SimpleVirusNet model. The above models shown in the Instance Template Model are visually developed using CoSMo. The tree structure of the entire model (including the specialized GernFastVirus and GenrSlowVirus models for the GenrVirus model) with multiplicity of model components are shown in Figure 3.

All operations including creation, deletion, and modification of simulatable and non-simulatable model components are supported visually. Similarly, adding ports, variables names of input and output values, and state variables are also supported visually. Other operations are complexity measure calculation and viewing of the generated source files and non-simulatable models. These un-timed non-simulatable models are simple and complex data structures and objects that are used in addition to the simulatable primitive and composite models [11].

3. CoSMoS ENVIRONMENT

The unified modeling and simulation environment bridges the CoSMo and DEVS-Suite by introducing visual tags for input and output ports of the models that can be developed in CoSMo. The visual toggling of input and output ports for tracking during simulation is advantageous as it eliminates the need to use dialogue boxes that are otherwise required for DEVS-Suite. Of particular importance is the manipulation of model components visually for designing experiments and on-the-fly selection of simulation data to be observed. In the following sub-sections, we describe the basic concepts and design that were developed for developing CoSMoS. The developed approach can be applied to other modeling approaches and simulation engines. For example, the rules for creating primitive and composite models can be defined according to Simulink block (discrete-

time and continuous-time) models or cellular automate models. The basic approach described in this section can be used to integrate solvers for these kinds of models into CoSMo and thus have other variants of CoSMoS.

A very simple conceptualization for the CoSMoS environment is shown in Figure 4. This integrated environment enables visual model development, model configuration and automatic simulation data collection, and simulation. It extends the CoSMo design and implementation to support visual configuration of models for experimentations and generation of partial simulation code for DEVS-Suite. Once the behaviors of atomic models (i.e., the external, internal, confluent, and output functions) are completely specified, the DEVS-Suite simulator can be invoked to simulate coupled models. Using CoSMoS with its model development process, modelers can develop and simulate models in an integrated visual modeling and simulation environment.

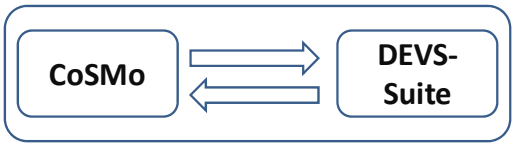


Figure 4. CoSMoS conceptual system view

3.1 Component Selection with Automated Data Collection and Observation

The models loaded in the DEVS-Suite are assigned default trackers. Users can select any coupled instance model and visualize any of its input and output ports as well as all of its components. For atomic models, common state variables (phase and sigma) and simulator timing variables (i.e., time of next event and time of last event) can also be visualized. Figure 5 shows the steps that lead to tagging input and output ports of models for tracking. (Recall that MFVC is the basic architecture of DEVS-Suite). The Controller object is responsible for the creation of the hooks with the View. The View object delegates the logic for determining the data for trajectory viewers through the TrackingControl object. Each tracker associated with the model has a checker that enables or disables what is to be tracked. With CoSMoS, instead of use dialogue boxes to select trackers, the user simply toggles on any port that is to be tracked. Figure 6 shows selection of the trackers visually for the net-virus network and experimental frame models. Note that the names (IDs) of the instance models are unique among themselves as well as the names of the instance template models. The background color of a port that is to be tracked during simulation is set to white by clicking on it.

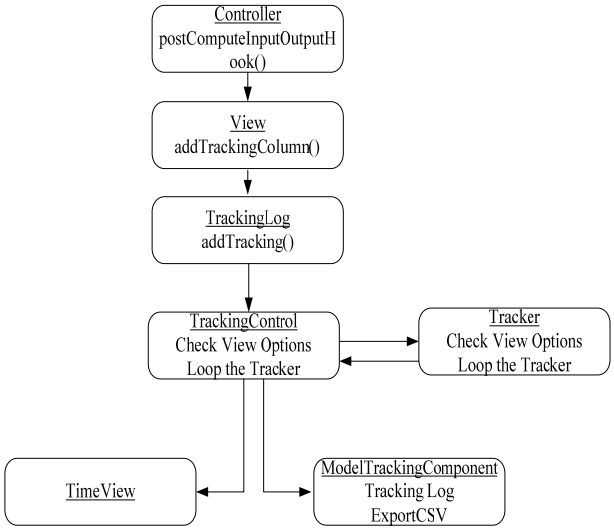


Figure 5. Selecting input and output ports for primitive and composite and their tracking in DEVS-Suite TimeView

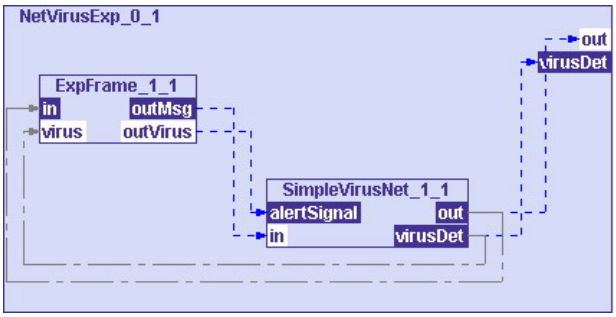


Figure 6. Input and output ports selected for tracking during simulation

3.2 Adding Behavior to Models

The primitive models that are transformed to DEVS atomic models must be completed before they can be simulated using the DEVS-Suite simulator. CoSMoS assists the user in adding behavior to generated source code. The structural specification of these Java models (e.g., input and output port names, variable types for messages, and skeleton code for transition and output functions) are automatically included in the generated source code. The source code for each model is consistent with its logical specification – i.e., name, ports, variables, and state variables included in the source code are the same as those that are stored in the database. Sample tabs of source code editors are shown in Figure 7. The editor is available as a part of the Netbeans editor API. Its functionalities include code coloring, line numbering, and keyword recognition. To disable the changes to the model's structure, the Guarded Sections property of the editor is used. The sections that are guarded cannot be edited. The guarded sections are shaded as shown in Figure 7. As noted earlier, guaranteeing that every logical model and its source code are consistent is crucial, otherwise verification of models and validation of simulations become unnecessarily difficult.

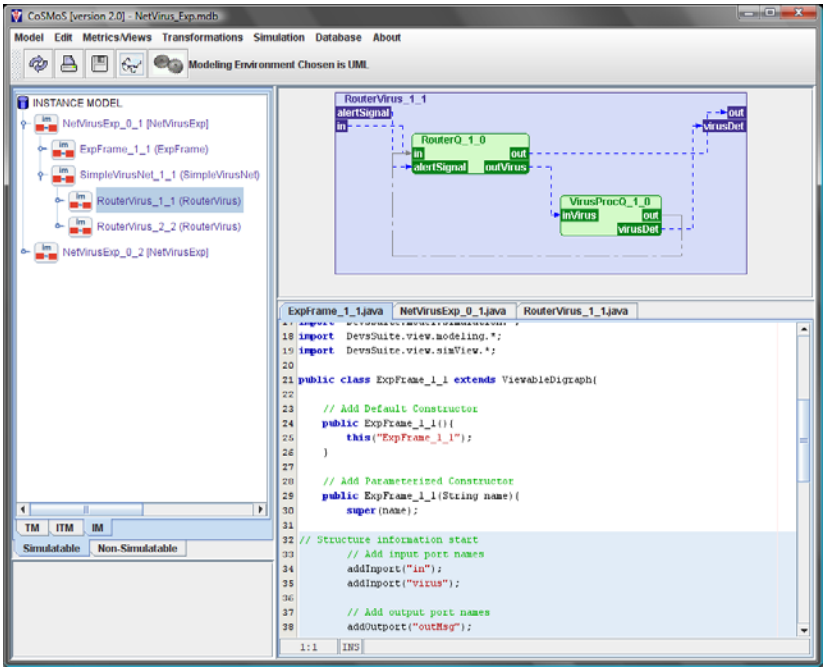


Figure 7. CoSMoS UI and the editor for adding behavior to models

3.3 Models and Namespaces

Another important need is to manage all models that are created and used in the CoSMoS environment. These models can be categorized into databases and flat files (see Figure 8).

A simple method is to devise a namespace. A root directory is defined within which any number of user-defined databases may exist in unique directories (e.g., NetVirus_Exp). Each database directory has a Database directory for databases (e.g., VirusNetwork.mdb), DEVS-Suite and XML directories, and a directory for NSM (non-simulatable) models. The DEVS-Suite Models directory consists of the Generated directory which has source and compiled files (e.g., VirusProcQ_0_0.java and VirusProcQ_0_0.class). As noted above, the source code for the primitive models (e.g., atomic DEVS model) must be completed in order to be simulated. The separation of the directories including the Generated and Compiled directories is useful for creating models for different users and/or systems to be modeled and simulated. The NSM Models directory has non-simulatable models. The remaining XML Models directory is designated for holding XML models).

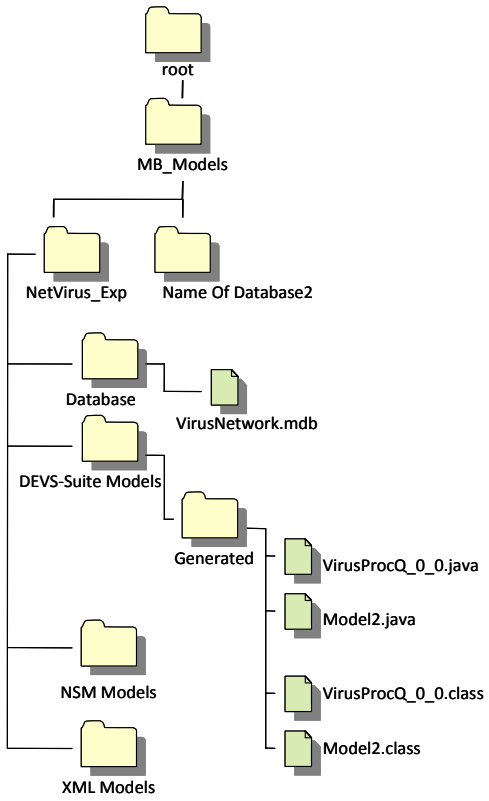


Figure 8. Namespaces for logical models and code for simulatable and non-simulatable models

4. CoSMoS PROCESS LIFECYCLE

The processes and relationships defined in Figure 9 are defined in terms of the following three parts. The Model Development activities are supported by CoSMo. The Experimentation Design and Configuration is supported by the capabilities described in Section 3. The model executions are supported by DEVS-Suite.

A. Model Development

1. **Select Database:** User selects the database that serves as a repository for the logical models. This database has a predefined structure (ER schema).
2. **Select an existing model or create a new one:** User uses an existing set of (partial or complete) models or creates an empty template model. User develops a family of models.
3. **Select Instance Template Model:** User selects one Instance Template Models from those that are created in Step 2. Other instance models may be created.
4. **Transform Instance Models into source code:** User instantiates one of the instance template models. For every specializee model, the user must choose one specialized model. User may create a family of alternative instance models based on the specialized models that are chosen. Then, user can generate partial and complete source code for all instance models of the selected instance model.
5. **Add behavior to the source code:** The primitive models are completed using the NetBeans editor. Other IDEs may be used, but the user must ensure the source code remains consistent with its counterpart logical model.

B. Experimentation Design and Configuration

6. **Select and load simulation models:** User selects an instance model to be simulated. The source code for the instance model and all of its components are loaded in the DEVS-Suite. The loading is an iterative process between completing the source code and automated compiling within DEVS-Suite.
7. **Select components and ports of models:** User selects models and their respective input and output ports. These selections are stored in the memory (JVM) in order to allow the user to select them for tracking (i.e., input/output trajectories, CSV export, and tabular form). The user may skip this step if no trajectories or tabulated data is to be viewed or no data is to be exported into a CSV file.
8. **Select visualization modes:** The modeler is given the choice of viewing the models' simulation output data on different types of trajectory viewers. The animation includes the SimView and the tracking of the output is shown in Tracking Log and TimeView.

C. Simulation Execution

9. **Execute model:** User starts simulation of the model. If any model component is selected to be viewed (see Step 7), the execution of the model is displayed as the animation of the model components, time-based trajectories, and tabulated form as well as exported CSV files for post processing, depending user's choice.

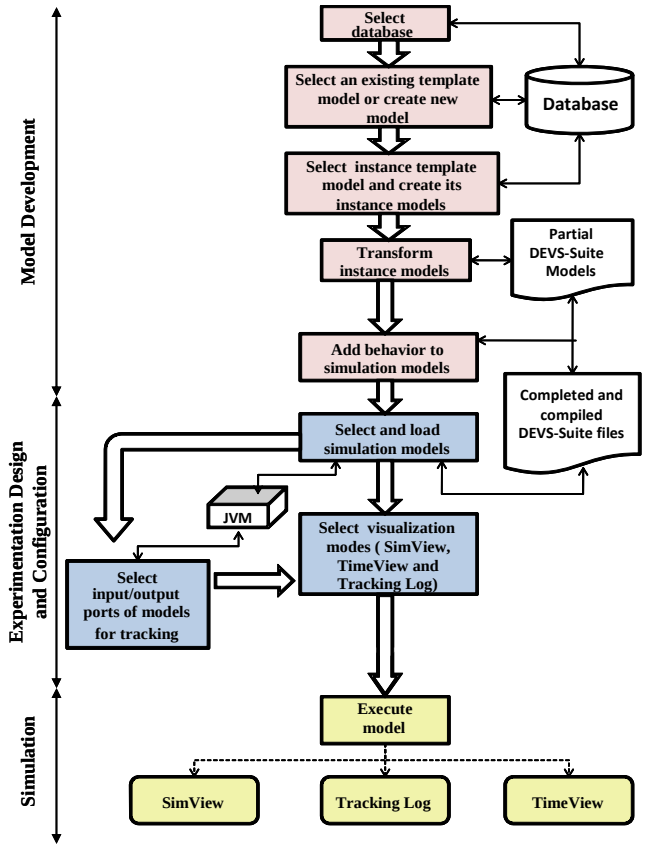


Figure 9. Process for developing models, designing experiments configuration, and executing simulations

5. COMPARISON WITH OTHER TOOLS

A variety of tools have been built for combined model development and simulation execution. Some tools support rendering source code of models as visual entities while few others support visual model development, chiefly through use of predefined icons or block component notations. Here, we focus on visual model development and selecting which of them to be monitored during execution. The behavior of the model components can be animated (view state changes of the components and input and output messages that are exchanged). The input and output data can also be viewed as time-based output trajectories.

Academic tools such as CoSMoS and Ptolemy II and commercial tools such as SimEvents support component-based modeling and simulation. They are aimed at different goals and differ from one another in important ways. Here we consider their visual modeling capabilities across modeling and simulation lifecycle. Ptolemy II [7] is a software framework developed as a part of the Ptolemy Project. It provides a component assembly framework which has a graphical user interface called Vergil. The project supports modeling and simulating of systems (e.g., real-time and embedded systems). It has a large domain-polymorphic component library. Its visual modeling offers pre-defined symbolic representation that can be assembled to create hierarchical models. The animation feature highlights the active model at any given instance of time during the simulation. The simulation results can be monitored and analyzed with the help of the pre-built plotters. The plotters form part of the model layout and thus can significantly increase the total number of the components that are displayed to users.

SimEvents is an extension of Simulink [8] and has a discrete-event simulation with a built-in model of computation. SimEvents interacts with the time-based dynamics of Simulink. It also provides signals or entity changes to control the processing of Stateflow changes. SimEvents can be used to develop activity-based models to monitor system states such as congestion, resource contention and processing delays. It provides pre-built libraries for queues, servers, switches, gates, timers, time-outs, and generators for entities, events, and signals. The SimEvents Sinks Library has several plotters that can be used in the models to monitor the values or the states of the various events. These sinks are strongly typed. Similar to Ptolemy II, the total number of components for a model can be very large depending on the number of input and outputs that are to be monitored.

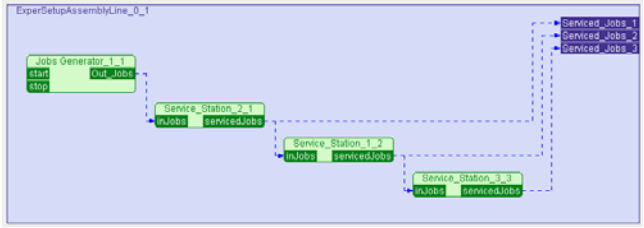


Figure 10. Assembly Line model in CoSMoS

To compare SimEvents, Ptolemy II, and CoSMoS, we developed a simple Assembly Line model as shown in Figure 10. This model is also developed in SimEvents and Ptolemy II (more details can be found in [4]). As shown in Table 1, since in CoSMoS, there is no need to include "monitoring components", the total number of components that are displayed to a user is always minimal compared with SimEvents and Ptolemy II. For a model with even a modest number of components, significant display space is required as compared with CoSMoS. Furthermore, while CoSMoS can support well-formed visual modeling of a family of models, these and many other tools must rely on a file structure provided by an operating system. In particular, the specialization relationship between a specializee and its specialized models are not defined in the file structures of operating systems. Therefore, users need to define their own scheme of organizing and managing a family of models for a system, something that is undesirable.

The Assembly Line model was simulated using CoSMoS, Ptolemy II, and SimEvents to evaluate the execution speeds of their simulators. For the Assembly Line model, the execution speed for Ptolemy II and DEVS-Suite are comparable and faster than the speed of SimEvents. With regard to the speed plotting of the trajectories, Ptolemy II and SimEvents performance can be much more efficient that DEVS-Suite depending on the number of plots and choice of programming language and implementation details.

Table 1. Visual display complexity metrics for component-based modeling tools

	SimEvents	Ptolemy	CoSMoS
Logical components	11	9	5
Ports	29	15	10
Couplings	14	11	4
Monitoring components	4	2	0
Trajectory viewers	4	2	4
Total no. of components	62	39	23

6. CONCLUSIONS

For studying complex and large-scale systems, it is desirable to have a unified modeling and simulation framework and tool that can reduce effort and help develop better models. Increasingly it is necessary to develop a family of models for a system and thus useful to enable not only developing models visually, automatically translating them to programming code, and making viewing of simulation of models simpler and more accessible to domain experts, but also to help automate management of multiple models of a system. The CoSMoS framework and its tool helps to address some of the challenges faced in developing models that are easier to verify and simulations that can be validated. As noted earlier, the key limitation for CoSMoS and all other tools that we are aware of is the inability for complete behavior specification. Toward greater support and ease, advances in domain-specific modeling are promising and could lead to a new generation of tools that can go beyond the current use of software engineering techniques and in particular depend on use of models with pre-fabricated behavior. Another interesting research direction for CoSMoS is to extend it to support common modeling approaches including cellular automata. The basic goal of the current and future research is to make modeling and simulation more accessible while strengthening the core verification and validation activities.

7. ACKNOWLEDGMENTS

This research is partially supported by NSF grant #BCS-0140269 and Intel Research Council.

8. REFERENCES

[1] Arizona Center for Integrative Modeling and Simulation. 2007. http://www.acims.arizona.edu.

[2] Bendre, S. and Sarjoughian, H. S. 2005. Discrete-Event Behavioral Modeling in SESM: Software Design and Implementation, Advanced Simulation Technology Symposium, p. 23-28, San Diego, CA, USA.

[3] Bradley, N. 2004. The XML Schema Companion, Addison Wesley.

[4] Elamvazhuthi, V. 2008. Visual Component-Based System Modeling with Automated Simulation Data Collection and Observation, Department of Computer Science and Engineering, Arizona State University: Tempe, AZ, USA. p. 1-115.

[5] Fu, T.-S. 2002. Hierarchical Modeling of Large-Scale Systems Using Relational Databases, Department of Electrical and Computer Engineering, University of Arizona: Tucson, AZ, USA. p. 1-114.

[6] Kim, S., Sarjoughian, H. S. and Elamvazhuthi, V. 2009. DEVS-Suite: A Component-based Simulation Tool for Rapid Experimentation and Evaluation. Spring Simulation Multi-conference, San Diego, CA, USA.

[7] Lee, E. A. 2003. Overview of the Ptolemy Project (No. UCB/ERL M03/25), Department of Electrical and Computing Engineering, University of California, Berkeley, USA.

[8] MathWorks, 2007. http://www.mathworks.com/.

[9] Mohan, S. 2003. Measuring Structural Complexities of Modular, Hierarchical Large-scale Models, Department of Computer Science and Engineering, Arizona State University: Tempe, AZ, USA. p. 1-112.

[10] OMG. 2004. Unified Modeling Language, http://www.omg.org/technology/documents/formal/uml.htm

[11] Sarjoughian, H. S. 2005. A Scaleable Component-based Modeling Environment Supporting Model Validation, Interservice/Industry Training, Simulation, and Education Conference, p. 1-11 Orlando, FL. USA.

[12] Sarjoughian, H. S. and Flasher, R. 2007. System Modeling with Mixed Object and Data Models. DEVS Symposium, Spring Simulation Multi-conference, Norfolk, VA, USA.

[13] Zeigler, B. P. and Hammonds, P. E. 2008. Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange, Elsevier.

[14] Zeigler, B. P., Praehofer, H. and Kim T. G. 2000. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Second Edition, Academic Press.