

VISUAL COMPONENT-BASED SYSTEM MODELING WITH AUTOMATED
SIMULATION DATA COLLECTION AND OBSERVATION

by

Vignesh Elamvazhuthi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

August 2008

VISUAL COMPONENT-BASED SYSTEM MODELING WITH AUTOMATED
SIMULATION DATA COLLECTION AND OBSERVATION

by

Vignesh Elamvazhuthi

has been approved

August 2008

Graduate Supervisory Committee:

Hessam Sarjoughian, Chair
Stephen Yau
Hasan Davulcu

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Many complex systems can only be studied using dynamical models that can be simulated. Models must have precise structural and behavioral abstractions in order to be correctly simulated. A key challenge in developing and executing simulation models is to have a modeling and simulation environment where users can systematically transition from model creation to simulation experimentation and evaluation. In response, this thesis develops a novel approach for component-based system modeling and simulation. An integrated modeling and simulation tool called Component-based System Modeling and Simulation (CoSMoS) is developed. Its modeling engine supports logical, visual, and persistent model specification with support for automated simulation code generation. Its simulation engine supports visual experimentation configuration and run-time data collection and observation. The CoSMoS tool enables simulation-based system design process with support for model verification and simulation validation. The integrated model specification, simulation code generation, and controlled experimentation capabilities of the CoSMoS tool are demonstrated with a model of an Anti-Virus Network software system.

To my parents

ACKNOWLEDGMENTS

First of all I would like to offer my sincerest gratitude to my thesis advisor Dr. Hessam Sarjoughian for introducing me to the research area of modeling and simulation. His patience and knowledge throughout the course of the research has been invaluable for the successful completion of the thesis. His professional guidance and encouragement helped me learn and appreciate the challenges of the subject.

I would like to thank my thesis committee Dr. Stephen Yau and Dr. Hasan Davulcu for their time and efforts in reviewing this thesis. I also would like to thank Dr. Yinong Chen who served as alternate during my defense examination.

I would like to thank all the members at ASU-ACIMS (Arizona Center for Integrative Modeling and Simulation), their help in discussions on my research, their support and friendship.

Finally I express my wholehearted thanks for my parents for their endless love and blessings. Their constant encouragement and support helped me to pull up my spirits when they were down.

TABLE OF CONTENTS

	Page
LIST OF TABLES	ix
LIST OF FIGURES.....	x
CHAPTER	
1 INTRODUCTION	1
1.1 Research Objective and Approach.....	2
1.2 Contribution	4
1.3 Organization of the Thesis	4
2 BACKGROUND	6
2.1 Component-based System Modeler (CoSMo).....	6
2.2 DEVS-Suite	10
2.3 Verification and Validation	15
2.4 Federation Development and Execution Process (FEDEP).....	16
2.5 Related Work	19
2.5.1 Ptolemy II	19
2.5.2 SimEvents.....	20
2.5.3 DEVS-Suite	21
2.5.4 Assembly Line Model Exemplar	21
3 COSMOS REQUIREMENTS SPECIFICATION	28
3.1 Preliminaries	28

CHAPTER	Page
3.1.1 Model	28
3.1.2 View	30
3.1.3 Control	32
3.2 CoSMoS Requirements	33
3.2.1 Instance Model Creation	34
3.2.2 Loading Models for simulation.....	36
3.2.3 Visual Component and Port Selection.....	39
3.3 CoSMoS Process Lifecycle.....	42
3.4 Verification and Validation using CoSMoS	45
4 COSMOS DESIGN AND IMPLEMENTATION	47
4.1 Instance Model Creation	48
4.1.1 Algorithms	51
4.1.2 Class Diagram	54
4.1.3 Sequence Diagram.....	55
4.1.4 Entity Relationship Changes	59
4.2 Export Models	63
4.2.1 Exported Models File Structure	63
4.2.2 Model Namespace	65
4.2.3 CoSMo Editor.....	66
4.3 Visual Model Component and Port Selection	67
4.3.1 Class diagram	68

CHAPTER	Page
4.3.2 Sequence Diagram.....	70
4.4 Loading Models for Simulation	73
4.4.1 Class Diagram	74
4.4.2 Sequence Diagram.....	75
5 DEMONSTRATION.....	77
5.1 Anti –Virus Model Example	77
5.1.1 Select Database or Create New Database	78
5.1.2 Model Creation.....	79
5.1.3 Create Model Instance.....	86
5.1.4 Adding Behavior	88
5.1.5 Configuration.....	90
5.1.6 Simulation.....	91
5.1.7 Simulation Results.....	92
6 CONCLUSION AND FUTURE WORK	94
6.1 Conclusion	94
6.2 Future Work.....	95
REFERENCES.....	97
APPENDIX A.....	100

LIST OF TABLES

Table	Page
1 Ptolemy II model components.....	23
2 SimEvents model components	25
3 DEVS-Suite model components	25
4 Comparison of visual complexity metrics	26
5 Relational Database Schema Specification for ModelClass Table	60
6 Relational Database Schema Specification for ExportTempTable Table.....	61
7 Primitive models in the Anti-Virus Model	80
8 Composite models in the Anti-Virus Model.....	81

LIST OF FIGURES

Figure	Page
1. CoSMoS Integration	3
2. Logical, visual, and persistent model types with model translators.....	6
3. CoSMo Client Server Architecture	8
4. SESM GUI.....	10
5. Architecture of DEVS-Suite (adapted from (Singh & Sarjoughian, 2003)).....	12
6. DEVS-Suite GUI	15
7. CoSMo and FEDEP.....	19
8. Assembly Line model.....	22
9. Assembly Line model in Ptolemy II	23
10. Assembly Line model in SimEvents.....	24
11. Assembly Line model in CoSMo.....	27
12. Instance Model Creation – The process.....	34
13. Loading models for simulation	37
14. DEVS-Suite Simulation Controls in CoSMo	39
15. Visual Configuration of Models.....	40
16. Visual Configuration of Models for Data Collection – The process	41
17. Process for creating and simulating models.....	42
18. Unique Instance Model creation – The Algorithm.....	52
19. Model-Class Relationship – The Algorithm.....	53
20. Class Diagram – Adding Instance Models	54

Figure	Page
21. Sequence Diagram – Select specialization	55
22. Sequence Diagram – Creating Instance of the Primitive Model.....	56
23. Sequence Diagram – Creating Instance of the Composite Models.....	58
24. Entity Relationship Changes	62
25. A sample of a generated atomic model.....	64
26. A sample of a generated coupled model	65
27. File-Directory Structure.....	66
28. Data Flow in DEVS-Suite	67
29. Class Diagram – Visually selecting components.....	68
30. Class Diagram – TrackingControl and Tracker.....	69
31. Sequence Diagram – Selecting input ports for tracking by mouse clicks.....	70
32. Sequence Diagram – Loading Models for Simulation	71
33. Sequence Diagram – Enabling models for tracking	72
34. Class Diagram – Components for loading models	74
35. Sequence Diagram – Loading models for simulation	75
36. Selecting existing database.....	78
37. Creating a new Model Template	79
38. Adding specializations to a model	81
39. Adding components to a model.....	82
40. ITM view of the models	83

Figure	Page
41. Adding input ports to a model.....	84
42. Adding couplings between two ports.....	85
43. Adding a state variable to a model.....	86
44. Creating Instance Model.....	87
45. Choosing specialization for a specialized model.....	87
46. Adding behavior to an already exported model.....	89
47. Selecting ports to track	90
48. Loading models for simulation	91
49. Options to select the output trajectory viewer	92
50. Class diagram of NBEditor implementation.....	100
51. A file generated by CoSMoS as seen in the editor	102

1 INTRODUCTION

Design and engineering of software-based systems remain an active area of research. Software architecture has a central role in building complex, large-scale software systems since it reduces development time, increases quality of detailed design and provides a holistic description of a system's specification (Medvidovic, Rosenblum, Redmiles, & Robbins, 2002). Modeling is needed to define and analyze the structural and behavioral aspects of a system. A general theory of Modeling and Simulation (M&S) was derived from the basic systems theory and thus provides a basis towards the engineering of software-based systems. Creating simulation models for systems can support developing designs that can be executed in virtual settings. These simulation models complement UML (Unified Modeling Language) models that are commonly used for software analysis and design (Ferayorni, 2008; Ferayorni & Sarjoughian, 2007). Simulation of software designs can support model verification and validation capabilities beyond what is generally supported by UML (Mooney, 2008).

DEVSJAVA (Arizona Center for Integrative Modeling and Simulation, 2007), an M&S tool implemented in JAVATM, establishes an environment that supports characterizing the models in DEVS (Discrete Event System Specification) (Zeigler, Kim, & Praehofer, 2000) formalism. The partitions in the architecture of the tool clearly delineate a modeling engine that realizes the logical DEVS modeling artifacts, and a simulation engine that realizes the parallel DEVS abstract simulator. The absence of the facility to automatically track model states and input/output trajectory makes it inapt for setting up experiments with hundreds of distinct models. The above mentioned capabilities have been introduced to the DEVS-Suite (Kim, 2008; Kim, H. S. Sarjoughian, R. Flasher,

& V. Elamvazhuthi, in preparation), an environment that extends the DEVJSJAVA Tracking environment (DTE) (Sarjoughian & Singh, 2004; Singh & Sarjoughian, 2003) with time-based data trajectories, tabular data, and CSV files. However DEVS-Suite does not support visual model development.

CoSMo¹ (Component-Based System Modeler) is a logical, visual, and persistent modeling framework that supports specification of models using a generic component-based paradigm (S. Bendre, 2004; S. Bendre & Sarjoughian, 2005; Fu, 2002; Mohan, 2003; H. S. Sarjoughian, 2005; Sarjoughian & Flasher, 2007). CoSMo supports specifying a family of models, where their scalability and complexity can be managed in a controlled manner. Given simulation engines such as DEVJSJAVA, models created in CoSMo can be mapped into partial simulation code.

1.1 Research Objective and Approach

The primary goal of this research is to integrate CoSMo and DEVS-Suite environments. The resulting environment *Component-based System Modeling and Simulation* (CoSMoS) is aimed at supporting development and configuration of simulation experiments using CoSMo's logical, visual, and persistent modeling engine specialized for DEVS models and can be executed using DEVS-Suite with automatic data observation and collection. The capabilities of the CoSMoS environment are:

- Visual selection of hierarchical model components for tracking.

¹ The name CoSMo is coined as a replacement for SESM/CM (Sarjoughian, in preparation). The new name captures more strongly the component aspect of system modeling.

- Follows a process for creating models and simulating them to conduct experiments.
- Display automatically gathered simulation data using a set of complementary data viewers.

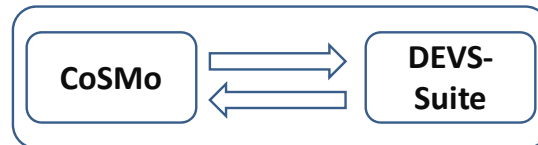


Figure 1. CoSMoS Integration

The overall approach to the integration was to observe the differences in the format of the models that are generated by CoSMo and can be simulated by DEVS-Suite. The version 1.3.0 of CoSMo provides the capability for creating, modifying, and deleting the structural aspects of the primitive and composite models (ACIMS, 2007). These structural aspects involve the name of the models, input/output ports (port names and data variables), couplings and modular hierarchical structures, multiplicity of model components, and specializations. CoSMo has a relational database for storing and managing the primitive and composite model types. CoSMo also supports some behavioral modeling (inputs, outputs, and states). This makes it suitable for the development of a family of models.

DEVS-Suite is an object-oriented modeling and simulation environment with the capability to track input, output, and state data sets. The models are described based on the system-theoretic modeling concepts and implemented in JAVA. The simulation models are syntactically checked for conformity by the Parallel DEVS simulator. The logical model to simulation code translator in CoSMo generates files that conform to the DEVS-Suite syntax and semantics. To enable simulation of these models in DEVS-Suite, a visual

modeling to simulation approach has been designed and developed. This involves completing the partially generated models in CoSMoS and loading them into DEVS-Suite. Using CoSMoS, modelers can develop and simulate models in an integrated visual modeling and simulation environment.

1.2 *Contribution*

The contributions of this thesis can be summarized as

- Extended CoSMo design and implementation to support visual configuration of models for experimentations and generation of simulation code for DEVS-Suite.
- Defined a process where model development and simulation can be carried out systematically.

1.3 *Organization of the Thesis*

Chapter 2 gives an overview and background of the CoSMo and DEVS-Suite environments being integrated. It involves the detailed description of the visual modeling engine CoSMo and its current capabilities. It also has a detailed architecture of the DEVJSJAVA Tracking Environment that is the basis of the DEVS-Suite. The related work discusses and compares the discrete event modeling and simulation environments Ptolemy II (Lee, 2003), SimEvents (*MathWorks*, 2007), and DEVS-Suite. The concept of Federation Development and Execution Process (FEDEP) is described with respect to the CoSMoS environment.

Chapter 3 describes the conceptual design of the CoSMoS environment. A process flow has been defined and explained in detail. It also discusses the necessary additions and

modifications required to facilitate the integration of CoSMo and DEVS-Suite environments. All the new or modified capabilities of CoSMoS are described in terms of use case diagram and basic requirements.

Chapter 4 shows the design for these capabilities which involves both class and sequence diagrams. The algorithms for the new capabilities have been described in detail along with a set of new database queries.

Chapter 5 depicts an example of Anti-Virus Model developed in CoSMoS. A set of models are developed step-by-step in order to show the capabilities of the CoSMoS.

Chapter 6 discusses conclusions and future research.

2 BACKGROUND

2.1 Component-based System Modeler (CoSMo)

Component-based System Modeler (CoSMo) is a modeling framework aimed at characterizing a family of system specifications. It defines a novel unified foundation for specifying *logical*, *visual*, and *persistent* primitive and composite models. Based on the concepts of modularity and (part-of and is-a) hierarchy, complex structures can be specified by coupling components' input and output ports. CoSMo supports component-based modeling approaches such as DEVS and XML (Sarjoughian & Flasher, 2007) which will be discussed in detail later.

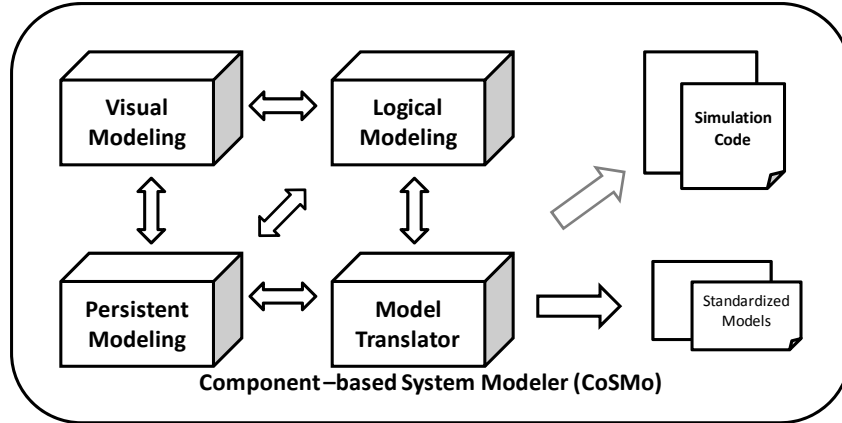


Figure 2. Logical, visual, and persistent model types with model translators

The logical model specification is governed by a set of axioms that ensure consistency among a family of alternative hierarchical model specifications. The models can have arbitrary complex *part-of* and *is-a* relationships giving rise to a large number of digraph (i.e., strict hierarchal) models. The persistent feature helps the modelers create, store, access and manipulate the models efficiently. The advantages of storing the model in a database include the management and scalability of models and being able to compute their complexity metrics. The visual modeling supports developing and manipulating large

models. CoSMo provides the facility to design the models at different levels of details due to the separation of the Template Models (TM), Instance Template Models (ITM) and Instance Models (IM). The CoSMo's translator supports transforming the logical models that are stored in the databases to their equivalent simulation code as well as other representations such as DTD (Document Type Definition) and XML (Extensible Markup Language) (Sarjoughian & Flasher, 2007). Logical models can be translated to simulation models. For example, for models that comply with the DEVS formalism and are intended to be executed with the DEVS-Suite, a translator has been developed. These DEVS-compliant logical models transformed to DEVS-Suite simulation code can be executed by adding functions that operate on inputs and state changes to produce outputs based on the given timing function. Translators have also been developed to generate DTD and XML models. The CoSMo models are explained in a more elaborate way using the state variables, ports, and the couplings that exist between the various models when the coupled models come into focus.

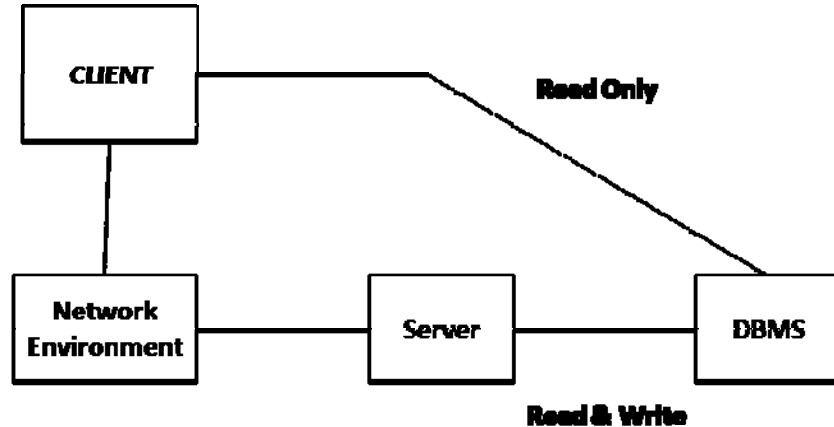


Figure 3. CoSMo Client Server Architecture

The basic architecture of the CoSMo is client-server (see Figure 3). The main parts of the software are Client, Network, and Server. The Client requests for the write requests, which are managed by the Network and then processed by the Server. The server also enforces rules according to the CoSMo's axioms in order to maintain the syntactical correctness of the models. All the read operations are directly handled by the database. The graphical user interface is efficient and provides three complementary views of every model: the Template model (TM), Instance Template Model (ITM), and the Instance Model (IM). The models are shown in the GUI by two means, one is using the Tree structure that lists all the primitive and composite models and their parts and specializations and the other is the block representation that shows the primitive models and the composite models up to two levels of its hierarchy. Using the DEVS-Suite translator, the CoSMo's primitive and composite models can be translated into partial DEVS atomic and complete coupled simulation code which can then be run by the DEVS-Suite simulation engine once it is completed. The partial DEVS-Suite source code generated for each atomic model can

be completed by providing the implementation of the external, internal, output, and time advance function templates using any IDE or the editor that is provided with CoSMo.

CoSMo also supports a class of Non-Simulatable Model (NSM) components. These types of models are based on object-oriented and XML model components. They are depicted differently than the Simulatable Model (SM) components which are time-based. The main difference between SM and NSM from a CoSMo perspective is that the execution of the SM model components is determined by the simulation protocol which generates simulation code. For example, if the DEVS-Suite simulator is used, then the Simulatable Models are executed according to the Parallel DEVS protocol.

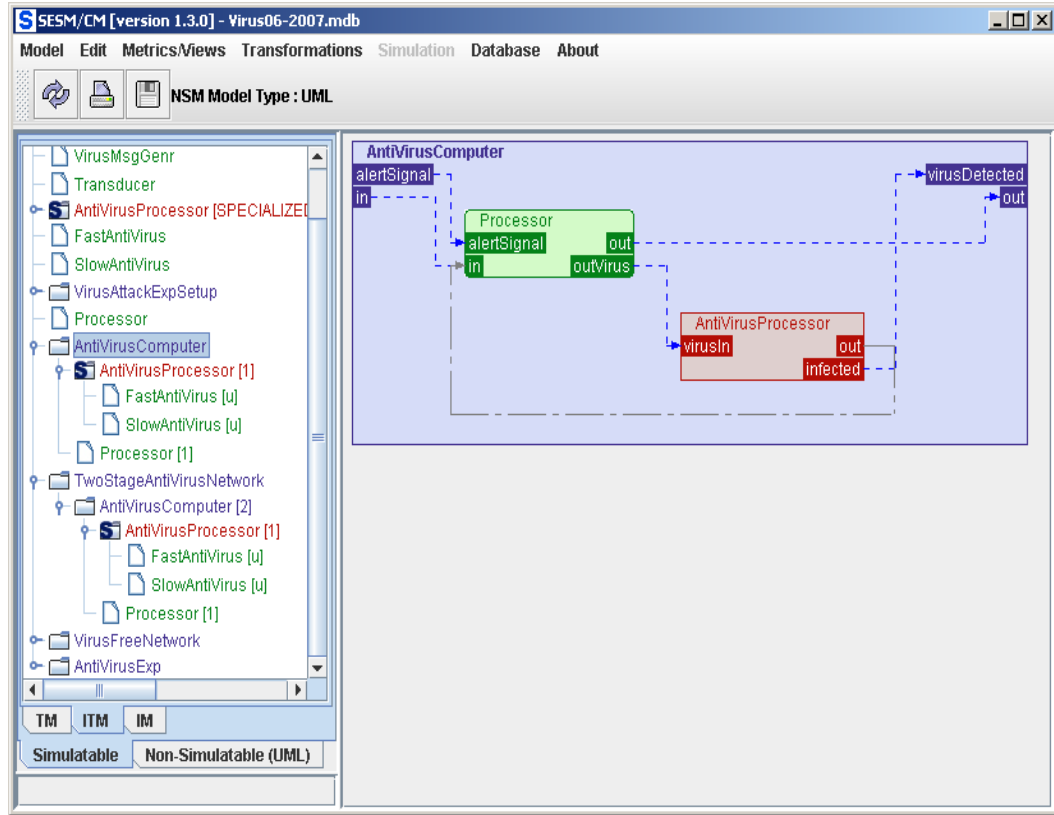


Figure 4. SESM (Scalable Entity Structure Modeler) GUI

2.2 DEVS-Suite

DEVS-Suite (Kim, 2008; Kim, et al., in preparation) extends the DEVJSJAVA Tracking Environment (DTE) (Sarjoughian & Singh, 2004; Singh & Sarjoughian, 2003). DTE is an object-oriented DEVS simulation environment. The models are syntactically checked for the conformation to Parallel DEVS. These models are simulated with the DEVJSJAVA simulation engine, an implementation of the DEVS abstract atomic and coupled simulators.

This simulation environment is comprised of a set of packages that support developing DEVS models. Two basic packages are the *devs.model.environment.modeling*

and *devs.model.environment.simulation*. The former supports a realization of the atomic and coupled modeling constructs. The latter is a realization of the abstract atomic and coupled simulators. The simulation model is typically defined as a set of instructions, rules, equations, or the constraints for consuming and producing input and output events. The models are designed with the Internal and External transition functions, time advance function, and output generation function to accept the input trajectories and thus generate the output trajectory over a period of time.

DTE is developed on strong system theoretic concepts and the classic MVC (MODEL-VIEW-CONTROL) design pattern. The details governing the modeling and simulation engines (MODEL) are strictly shielded from the VIEW and CONTROL.

The MODEL is independent of CONTROL and VIEW. The MODEL is processed under the directive of the CONTROL and the data is consumed by the VIEW. The CONTROL does not introduce any side effects to the MODEL. The CONTROL maps user actions to their counterparts provided in the MODEL (for example, injecting input to a model). The VIEW does not change the simulation models; instead it supports accessing simulation models input and output variables and common variables (i.e., sigma & phase; tL & tN) that belong to all models and the simulator. The VIEW provides an interface through which the simulated model can be executed under the DEVS-Suite execution scheme.

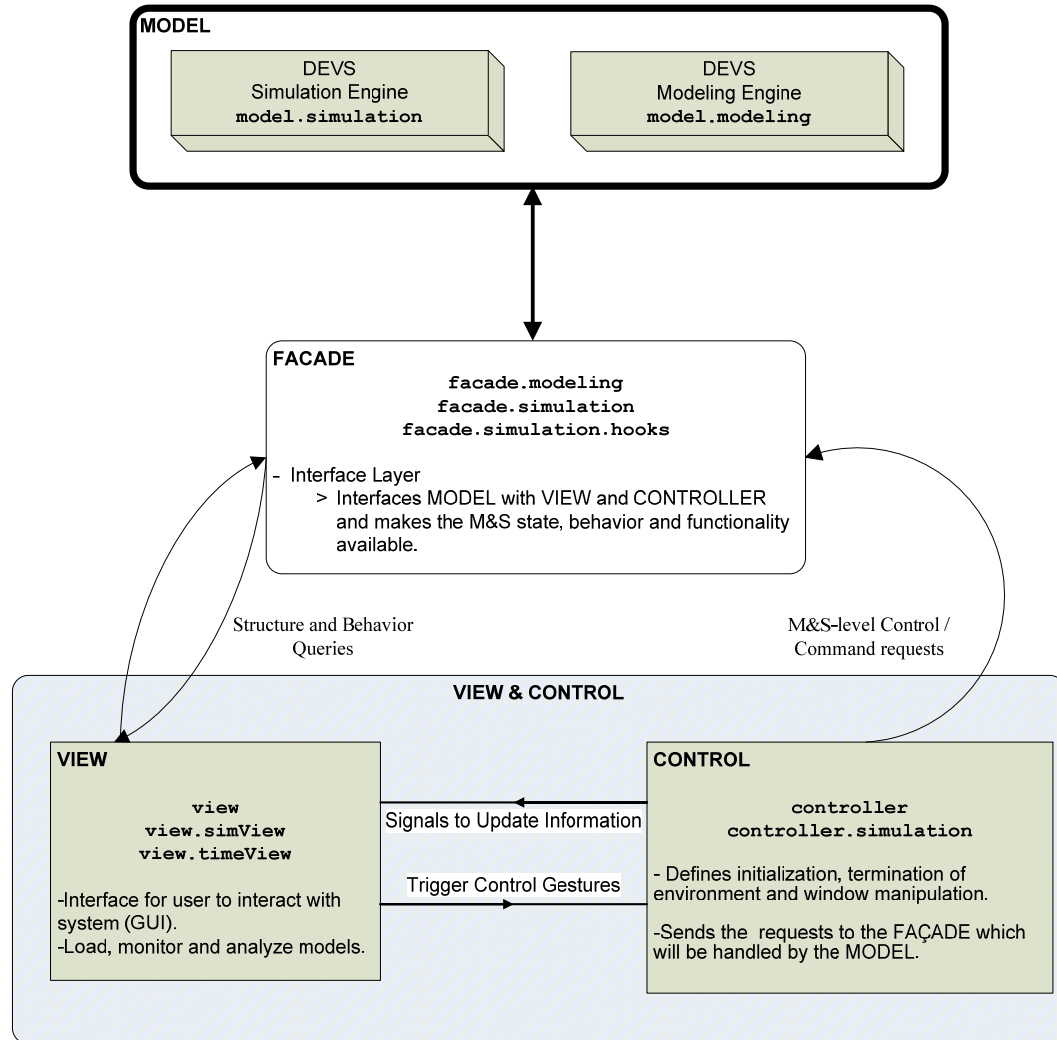


Figure 5. Architecture of DEVS-Suite (adapted from (Singh & Sarjoughian, 2003))

The architecture (Figure 5) of DEVS-Suite has modeling and simulation engines that are complex in nature and are treated as part of the MODEL of the MVC decomposition. The MODEL represents the atomic and coupled models. The Façade design pattern is used to expose inputs, outputs, and states of the models as well as simulation control operations. The FACADE manages all external VIEW and CONTROL

interactions of the MODEL. This FACADE Interface layer maintains a precise set of operations in such a way that the MODEL's internal details are invisible to the VIEW and CONTROL. For this layer to communicate with the VIEW and CONTROLLER, the Coupling and Communication (C&C) layer is introduced. The C&C layer has the simulator control logic built into it. The VIEW and CONTROLLER can send and receive data and control messages to the FACADE interface layer (and thus the MODEL) only through the C&C layer.

The VIEW serves as a visualization interface for the user to interact with the MODEL through the CONTROL. The VIEW displays some aspects of the simulation models to the user. It only has access to the information that is available from the Façade and C&C layers. The VIEW can be considered as a workspace to view, control, and monitor simulation models. It also orders all user interactions. However, there is no guarantee that the VIEW can display the data it receives from the C&C at the same rate the MODEL is generating them. This is because the VIEW does not control the execution of the simulation models (i.e., MODEL) and therefore pulls the data from the C&C layer independently of the CONTROL and MODEL.

The CONTROLLER defines the overarching execution logic which includes initialization, termination of environment, and VIEW and MODEL manipulation. The control requests are originated in the VIEW due to a user request or action. The CONTROLLER also defines proxies for the simulation engine's execution logic which are Reset, Run, Run[n], Inject, and Pause operations. The user also has the choice of

controlling the speed of the simulator and animation. These operations are managed through the C&C layer. When the logic for processing control request is not present in the model level logic, the CONTROLLER maps it into the corresponding section in the C&C layer.

The central feature of DEVS-Suite is to allow the user the option to select the components and thus observe only the input, output, and state variables that are of interest. This capability simplifies the configuration of different simulation experiments without adding auxiliary code to the simulation models or writing transducer models as is commonly done. This kind of setup helps in analysis by enabling the setup of simulations and therefore tracking the states, and input/output in the three complementary views (tabular, time trajectories, and animation) in a controlled and repeatable manner.

2.3 Verification and Validation

Verification refers to the process of analyzing the extent to which the model developed pertains to its requirements and specifications. Verification also evaluates the extent in which the model and simulation developed conforms to the established software

and systems engineering techniques. Validation refers to the process of analyzing the degree of similarity in the simulated model with the real (or imagined) system while conforming to the prescribed (or desired) structural and behavioral requirements.

A disciplined approach to the V&V of these simulation models can reduce developing and integration risk while enhancing the credibility of the simulations. The iterative nature of simulation model development in CoSMoS helps the modeler carry out modeling and simulation tasks systematically.

2.4 *Federation Development and Execution Process (FEDEP)*

High Level Architecture (HLA) has been defined to introduce interoperability among simulations and also reuse. Thus HLA enables various types of simulation (logical and real). HLA Object Model Template (OMT) plays an important role in building HLA-compliant simulations (Lutz, Scrudder, & Graffagnini, 1998). The HLA/OMT specifies two object models: Federation Object Model (FOM) and Simulation Object Model (SOM). A FOM deals with the issues of decomposition of federations into federates while a SOM deals with the dynamic capabilities of the federates, such as their operations to the extent of capturing interactions. There are two main technical objectives for HLA/OMT specifications. The first objective is to provide a common specification for the exchange of the data and coordination among the members of the federation using the concept of publish and subscribe. The second objective is to provide a common mechanism for describing the capabilities of potential federation. FEDEP defines seven basic steps for the HLA federations to develop and execute their federations. The steps are as follows:

Step 1. Define federation objectives

The federation user, sponsor, and the developer define and agree on a set of objectives.

Step 2. Perform conceptual analysis

Based on the characteristics of the problem space, a representation of the real world domain is developed.

Step 3. Design federation

A plan is developed for federation development and integration.

Step 4. Develop federation

The Federate Object Model (FOM) is developed.

Step 5. Plan, integrate, and test federation

Federation integration and testing is conducted to ensure the interoperability requirements are met.

Step 6. Execute federation and prepare outputs

The federation is executed and the output is pre-processed.

Step 7. Analyze data and evaluate results

The output data from the federation execution is analyzed and evaluated.

We can observe that there is a direct relationship between HLA FOM and SOM with DEVS. The atomic and coupled models correspond to federate and federation components (Sarjoughian and Zeigler, Simulation Transactions, 2000).

As seen in the FEDEP modeling and simulation life cycle, simulation model development and execution is an important component. To build conceptually correct models and correct simulation code for large scale complex systems, an environment should consider the following:

- Formal model specification: The logical models in CoSMo follow specific rules and axioms to support well defined (component-based) structure and behavior specifications.
- Visualizations: CoSMo allows the modeler to develop large and complex models. Visualization of the simulation output data is provided with the help of DEVS-Suite's viewers (e.g., time trajectories of inputs, outputs, and states).
- Repository: The models are stored in the database and thus are persistent across different sessions.
- Transformation: Models can be translated into simulation code for a class of simulation models (e.g., DEVS).

Since CoSMo's primitive and composite models can represent DEVS atomic and coupled models, CoSMo and DEVS-Suite can be used together to support model verification and simulation validation. As we see in Figure 7 (H. Sarjoughian, 2005), CoSMo supports four phases of the FEDEP (i.e., Develop Design, Develop Conceptual Model, Validate Conceptual Model, and Verify Design)

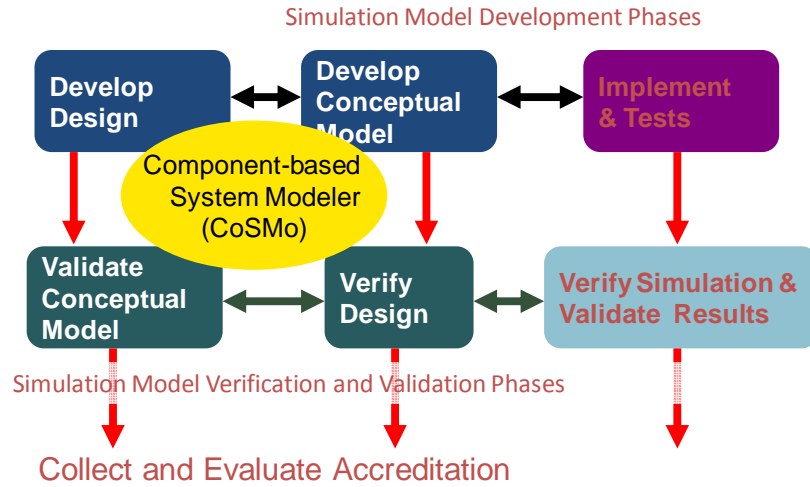


Figure 7. CoSMo and FEDEP

The integration of DTE in CoSMo allowed the modeler to implement the DEVS model and simulate them using the DEVS simulator available in DTE. The models can be structurally configured; however, for behavior specification the modeler needs to manually complete the models using the IDE available in CoSMoS. The simulation results of the models developed above can be shown in various output trajectory viewers available.

2.5 Related Work

2.5.1 Ptolemy II

Ptolemy II (Department of EECS, 2007) is a modeling & simulation framework developed as a part of the Ptolemy Project. It is a component based framework implemented in JAVA and has a graphical user interface called Vergil. The project aims at studying modeling and simulation of real time and embedded systems. It has a large, visual, domain-polymorphic component library. A component called *Director* defines the interaction semantics among a set of models and the director that is for discrete event

models is called *DE Director*. The models are pre-defined for a given domain and specific visual representations. These model parameters can be set visually, but changes to each model's logic (e.g., functions) must be done manually (i.e., through the use of text editors). The models can be visually coupled together. However, they are not auto-arranged and thus it is the responsibility of the modeler to manually adjust their positions. The animation feature shows one active model at any given instance of time during the simulation. These simulation results can be monitored and analyzed with the help of pre-built plotters. The plotters form part of the model layout and increases the number of the components in addition to the models that are simulated. The components used in Ptolemy II are domain specific and the modeler needs domain knowledge in order to use them.

2.5.2 *SimEvents*

SimEvents is an extension of Simulink which has a discrete-event model of computation built into it. SimEvents can be used to develop activity-based models to monitor system parameters such as congestion, re-source contention, and processing delays. It provides pre-fabricated 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 and thus use of an incompatible value at one of the ports will result in an error. SimEvents provides an environment for modeling hybrid dynamic systems containing continuous-time, discrete-event and discrete-time components.

SimEvents interacts with the time-based dynamics of Simulink. SimEvents also provides signals or entity changes to control the processing of State flow changes.

2.5.3 *DEVS-Suite*

DEVS-Suite is an environment targeted for simulating parallel DEVS models. It uses the DEVJSJAVA simulation engine and introduces the capability to configure input and output variables and predefined state variables for observation and data collection. Data can be viewed as time-based trajectories and in tabular form during simulation execution. DEVS-Suite use the Model-View-Control architecture as described in Section 2.2. DEVS-Suite supports simulating atomic and coupled model types. The atomic model contains input and output ports and variables, state variables and parameters, and time advance, internal, external, confluent transition, and output functions. The composite model defines the way in which atomic and/or components can be coupled together. However, there is no support for visual model development – i.e., template Java code must be completed using a text editor or IDE such as Eclipse. The input and output messages between the models can be animated and their state and parameters visualized during simulation execution. The models can be moved around manually in the simulation viewer, but the couplings are static and are relatively aligned. Due to this, they often overlap and reduce the visual clarity of the model.

2.5.4 *Assembly Line Model Exemplar*

The Assembly Line (Jayadev, 1986) model shown in Figure 8 is chosen to compare Ptolemy II, SimEvents, and DEVS-Suite simulation tools. Jobs are generated by a

Generator model at predefined intervals and are serviced by three processors P_1 , P_2 , and P_3 in a cascade fashion. The service time for each job is specified by a Processor.

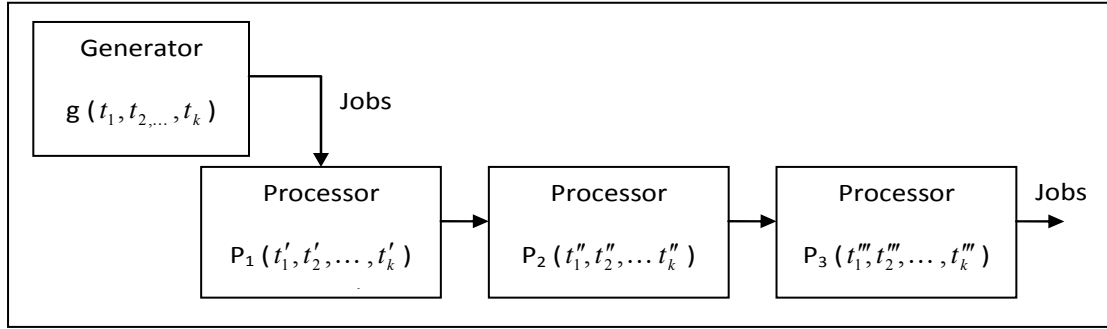


Figure 8. Assembly Line model

2.5.4.1 Observations

We considered Ptolemy II, SimEvents, and DEVS-Suite to analyze the visualization aspects of models and their simulations. The Assembly Line shown in Figure 8 is a model that generates jobs and processes them using multiple processors in a cascading fashion. This exemplar model is part of the demos bundled with Ptolemy II.

The components used in Ptolemy II to build the sample model are shown in Table 1. Several components have to be combined to represent a single entity. For e.g., pulse generator and *NonInterruptable* Timer form the processor used in the Assembly Line model. Visual monitoring components have to be added as a part of the model to observe the output and behavior of the model during the simulation. The models and the couplings must be adjusted manually to avoid overlapping and to enhance visual feedback. The layout of the Assembly Line in the Ptolemy II environment can be seen in Figure 9.

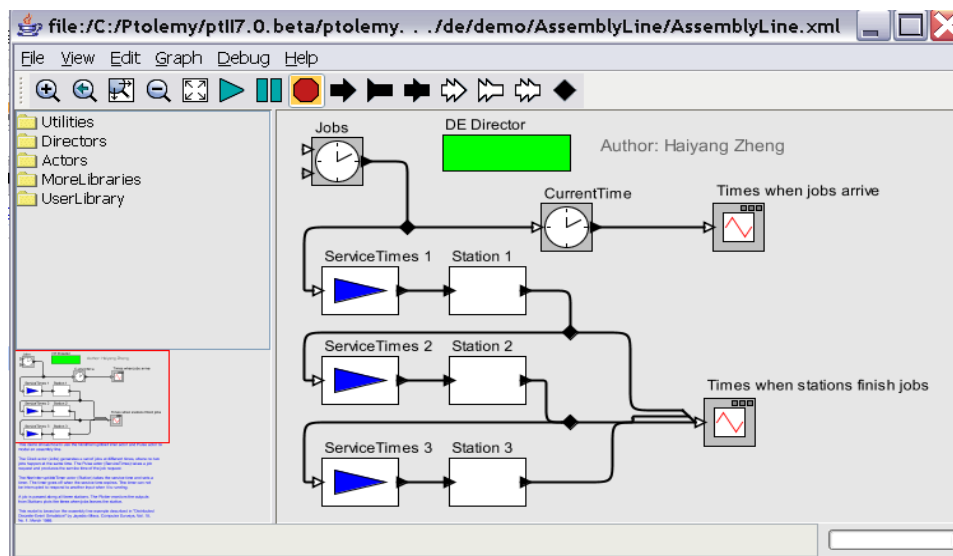
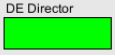


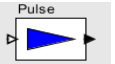
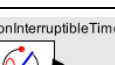



Figure 9. Assembly Line model in Ptolemy II

Table 1

Ptolemy II model components

	Component Name	Icon Representation	Name in Model
Ptolemy II	DE Director		DE Director
	Clock		Jobs
	CurrentTime		CurrentTime
	Pulse		ServiceTimes1, ServiceTimes2, ServiceTimes3,
	NonInterruptibleTimer		Station 1, Station 2, Station 3
	TimedPlotter		Times when jobs arrive, Times when stations finish jobs.

The Assembly Line model was also developed in SimEvents. The ports in SimEvents have to be manually adjusted. The ports are checked for types before they can

be coupled. The components have been categorized as logical and visual components, ports, and couplings. The visual components are the graphs and the plotters that capture the simulation data. These probes are shown as separate entities in the model layout. As the models are synthesized using basic components from libraries, some functionality, such as queuing of jobs in the server, needs the queue component to be added explicitly to the model. To track any particular component of the model these probes have to be added in addition to the models. Brief descriptions of the models are given in Table 2 and the layout is shown in Figure 10.

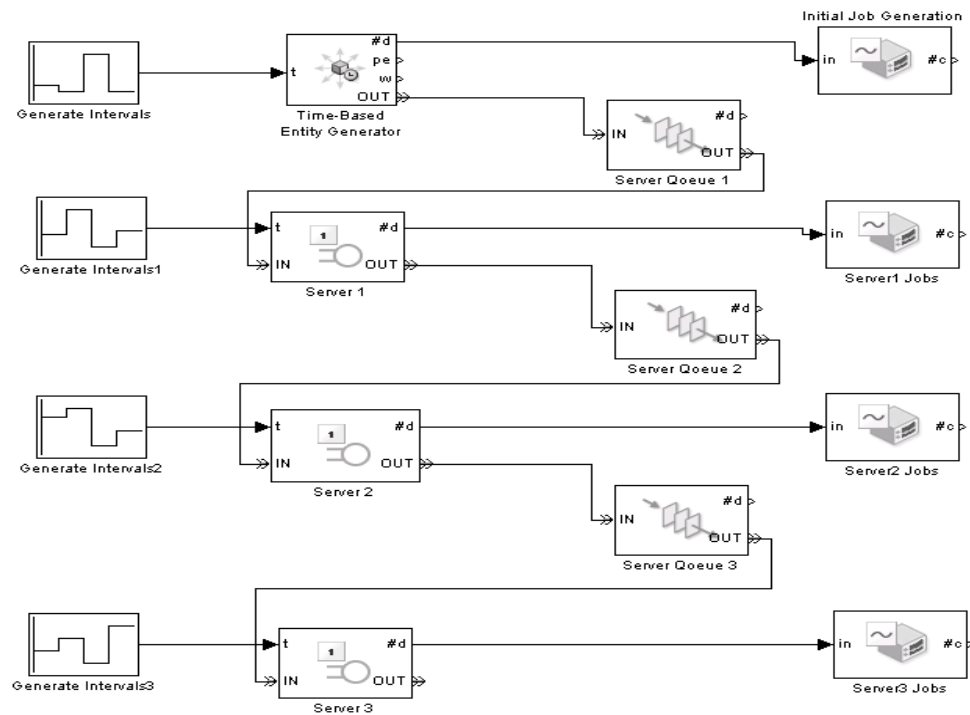

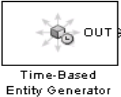
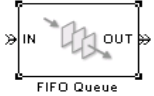



Figure 10. Assembly Line model in SimEvents

Table 2

SimEvents model components

	Component Name	Icon Representation	Name in Model
SimEvents	Event based sequence generator		Generate Intervals 1, 2, 3, 4
	Time-Based Entity Generator		Job Generator
	FIFO Queue		Server Queue1, Server Queue2, Server Queue3
	Single Server		Server 1, Server 2, Server 3

A brief description of the models and the components are given in Table 3. DEVS-Suite is a visual simulation tool where the models development is through code. The simulation viewer (Figure 6) shows the state information of each component during the course of the simulation. The ability to animate the messages passing between the models reduces need for additional visual monitoring components, ports and couplings associated with them.

Table 3

DEVS-Suite model components

	Component Name	Name in Model
DEVS-Suite	Atomic Model	Entity Generator
	Atomic Model	Service Station 1, Service Station 2, Service Station 3,
	Coupled Model	ExperSetup

Table 4

Comparison of visual complexity metrics

	Ptolemy	SimEvents	DEVS-Suite
Logical Components	9	11	5
Ports	15	29	10
Couplings	11	14	4
Monitoring Components	2	4	0
Trajectory Viewer	2	4	4
Total No. of Components	39	62	23

The visual complexity metrics of the Assembly Line model with respect to the different environments are shown in Table 4. The metrics reveal that as the scale of a model increases, the number of components would increase for Ptolemy II and SimEvents with respect to DEVS-Suite. From the table it can be observed that visual components are the major contributors to the overall visual complexity of Ptolemy II and SimEvents. In contrast, DEVS-Suite does not require components such as TimedPlotter; instead dialogue boxes are used.

Feedbacks were also added to observe the alignment of the models and their couplings. As already mentioned in Ptolemy II and SimEvents the components and couplings had to be manually adjusted. DEVS-Suite allows alignment of the model, but does not support couplings that cross over model components. Such overlaps and difficulty in adjusting the models makes it very challenging to manage for large-scale models.

The example model mentioned in Section 2.3.0.1 (i.e., the Assembly Line) has also been developed in the CoSMoS environment. After comparing the visual complexity of CoSMoS to that of DEVS-Suite, it can be seen that a number of logical components, ports

and couplings are similar. The advantages of CoSMoS over DEVS-Suite involve visual model development and not needing to use customized code or dialogue boxes for simulation data collection and observation.

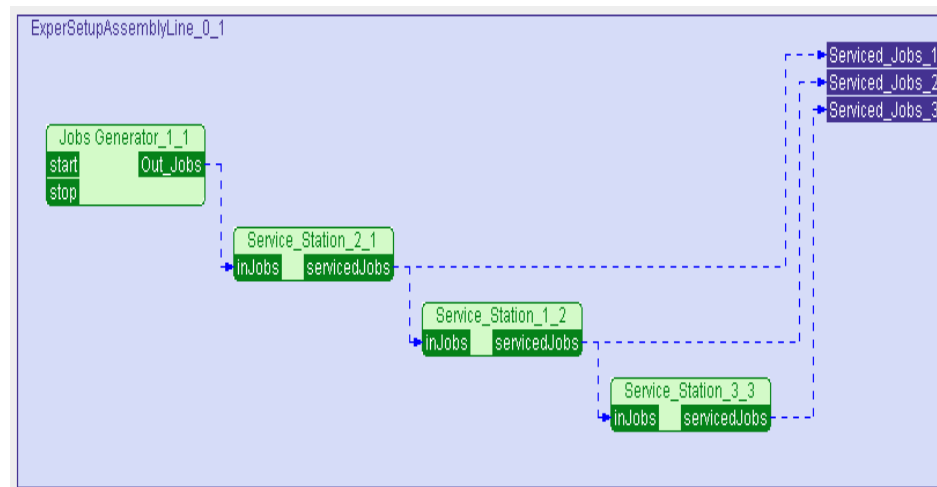


Figure 11. Assembly Line model in CoSMo

3 CoSMoS REQUIREMENTS SPECIFICATION

The major components for the integration, i.e., CoSMo and DEVS-Suite, were analyzed closely and the shortcomings from each were identified. An overall architecture for the integrated system CoSMoS (Component-based System Modeling and Simulation) has been designed. The components involved in the integration have been explained clearly with respect to the integrated architecture. The problems identified in both CoSMo and DEVS-Suite have been realized as requirements and have been described using a detailed use-case diagram. A detailed process flow has been defined for the integrated environment. The process flow shows the sequence of steps for the creation of visual models, adding behavior, and simulating the models using the DEVS-Suite controls.

3.1 Preliminaries

The design of the CoSMoS environment is based on the Model-View-Control (MVC) design pattern. Both CoSMo (S. Bendre, 2004; S. Bendre & Sarjoughian, 2005; Fu, 2002; Sarjoughian, 2001; Sarjoughian & Flasher, 2007) and DEVS-Suite (Kim, et al., in preparation; Sarjoughian & Singh, 2004; Singh & Sarjoughian, 2003) environments are developed using the principles of the MVC (Trygve M. H. Reenskaug). The Model, View, and Control components of CoSMoS are described next.

3.1.1 Model

3.1.1.1 Logical Models (CoSMo):

The primitive and the composite models are defined in CoSMo. Each of these models is represented as both Template Model and Instance Template Model. The Instance Template Models (ITM) can also be instantiated to Instance Model (IM) which shows the

realizations of the specializations and concretely defines the multiplicity of the sub-components if it is unspecified.

The primitive component corresponds to the Template Model(TM) or an IM. In the TM, the primitive model can be specialized using is-a relationship. The term specializee refers to the component that has a specialization relationship to the specialized models. The input/output interface of the specialized model is same as the interface of the specializee. However the state variables of two specialized models can be different.

A composite model corresponds to the TM, ITM or IM. The composite model consists of primitive or other composite model. It also has states, input/output ports, and a set of couplings between the ports contained within it.

3.1.1.2 ER Specification, Persistent Models (CoSMo):

The structural models in CoSMo are described and stored in a relational database in terms of structural features of the model components such as identity (i.e., model name), hierarchy (i.e., decomposition), input/output interface (i.e., port names), and their creation time. For the models to be executed there are some behavioral requirements which need to be added to the existing models. These are described in terms of port variables, state variables, and NSM variables.

3.1.1.3 Atomic & Coupled Models (DEVS-Suite)

The models in DEVS-Suite are based on the DEVS formalism which can be mathematically expressed. As mentioned earlier, there are two types of models: atomic and coupled. Atomic models are the basic models from which the coupled models can be built.

Atomic models have the ability to define behavior with time base, inputs, outputs and functions for defining the next states. The higher models, i.e., the coupled models, are composed of other atomic and/or coupled models connected to each other by couplings in a hierarchical manner. These models are supported with input and output ports to enable communication with each other and the outside world.

3.1.2 View

3.1.2.1 Hierarchical tree and block model representations (CoSMo)

The graphical user interface of CoSMo plays a major role in supporting visual model development. The hierarchical models are displayed in a structured tree format using the JTree format of JAVA. Although the couplings and ports are not visible in the tree structure, it is complemented by the block diagram layout with ports and couplings. The block diagram shows the models at two levels of hierarchy in a single display. The advantage of a well defined user interface of CoSMo was to streamline the process of developing models so that they can be developed in an orderly fashion, i.e., create the template models and then create the instance models from those. These visual representations are consistent with the model information in the database. The ports in the block can be selected to configure the model for simulation.

3.1.2.2 SimView (DEVS-Suite)

The SimView is a simulation viewer that has a visualization of the structure and behavior of the hierarchical DEVS models. The hierarchical and the component structure are derived from the source code written in JAVA that conforms to DEVS specifications.

The view uses a boxes-within-boxes visual metaphor to portray all of the components in a model and their position within the hierarchy. The individual input/output ports and their couplings with models in the current or different level of hierarchy are clearly shown in the viewer. It also shows the movement of messages as the simulation progresses. This helps in presenting the dynamics of the model and simulation in a detailed manner. The view also possesses capabilities to give inputs to models that can be defined on the ports in the simulation code.

3.1.2.3 Time-based and Tabular Trajectories (DEVS-Suite)

The simulation data selected for observation can be collected in the HTML table and observed at run-time. This data is retrieved from the DEVS-Suite Communication and Control layer and displayed to the modeler.

Time-based trajectories are displayed in X (value) and Y (time) coordinates. These coordinates represent observed data values, such as input events at a series of monotonically increasing time instances. The Y coordinates are single-valued and can be numeric or symbolic. The values show simulation time (or clock) which is determined by the simulation model's time advance function. The X coordinate can be either single- or multi-valued. The X coordinate may represent input, output, or state values of models. Models can be continuous, discrete-time, and discrete-event. That is, the semantics of the data displayed are based on the DEVS models. DEVS-Suite also supports separately assigning units to the X and Y coordinates (e.g., the unit of time in one plot can be seconds while the unit of time in another plot can be milliseconds).

To allow the TimeView to become a viewer of time-based simulation, the semantics of the data it displays are provided by the DEVS simulation engine. That is, the TimeView displays data it receives, but the correctness of input, output, and state trajectories are due to the models and their simulation.

3.1.3 Control

3.1.3.1 Visual Modeling Gestures (CoSMo)

The visual modeling gestures in CoSMo environment include creating, modifying, and deleting models. It supports modelers by specifying their models in an iterative and incremental fashion. The operations on the models are persistent, i.e., the models are stored in the database. The models follow a strict flow in their creation. The template models need to be created before they can be instantiated and are ready to be simulated. The structural specification for a model can be specified from the GUI at three places

- The menu bar defined at the top of the application.
- A pop-up menu showing options for the model on the tree structure. This menu can also be invoked from the graphical block representation of models. CoSMo also enables behavior specification of the atomic models. The behavior of these atomic models is specified in terms of input/output variables, state variables, and state transition functions. Currently CoSMo provides pop-up menus on atomic models to add input/output variables and state variables. The state transition functions can be added into the model using a built-in source code editor. There are several available.

3.1.3.2 Simulation Gestures (DEVS-Suite)

The simulation gestures from DEVS-Suite forms the part of the model-level logic that includes behavior such as simulation and model manipulation (Run, Pause, Step, Step (n), and Restart). The control gestures are usually triggered from the VIEW by a user as a task to be completed. The logic for the processing of the control request is not present in the model-level logic itself. Instead, a mapping of the request is sent to the Coupling and Communication layer.

These controls are loaded on the GUI of CoSMo once the model has been successfully loaded into the simulation engine.

3.2 CoSMoS Requirements

For the successful integration of CoSMo and DEVS-Suite environments there were several requirements that had to be met. In the following, we describe the requirements for the integrated CoSMoS environment given in each of the following use cases.

3.2.1 Instance Model Creation

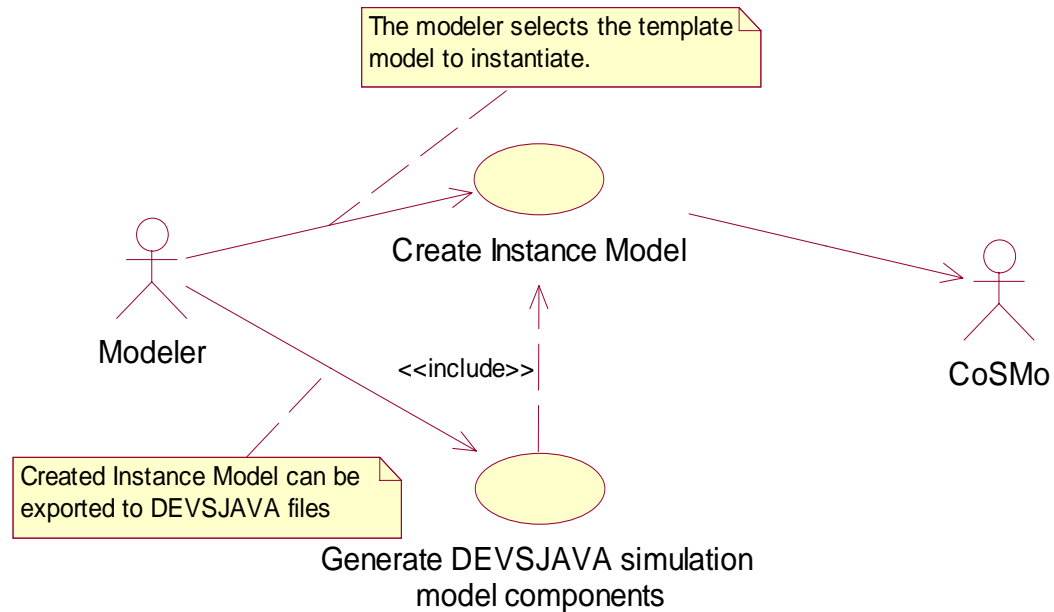


Figure 12. Instance Model Creation – The process

The use case in Figure 12 shows the instance model creation process in the CoSMo environment. The instance model forms the instantiation of the Template Model where the multiplicity of the components is explicitly identified and the specializations are reserved. The generation of the partially complete Java files is dependent upon the instance model creation. The process to create the Instance Model had to be enhanced to incorporate new capabilities, like identifying the corresponding simulation code of the selected model and reuse of an existing model with the same set of components.

The requirements identified to implement the new Instance Model creation method are:

- 1) Each model needs to be given unique names to distinguish between models with different multiplicity and specialization.

- 2) Root models instantiated to the same set of constituent components should not be allowed to be recreated.
- 3) The Components that are a part of the multiplicity in a model must be distinguishable between them once the instance has been created.
- 4) Instances of the component models as a part of a higher model should be reused if an instance of the same component with identical configuration exists
- 5) A clear mapping of the model names and their respective classes should be defined which would be needed for exporting the instance model to its respective JAVA file.

The requirements listed above were carefully analyzed and changes to the design were proposed in terms of a revised algorithm for the Instance Model Creation, a new algorithm to establish the mapping between the Models and the classes to be used and new additions to the database and existing program logic.

- 1) The instance model identifications were chosen based on the FCFS (First Come First Serve) basis. Any model can be either a root model or a constituent of another model; the instance identification numbers would be generated based on order of their creation.
- 2) Each and every model's instance and its constituents are checked for redundancy and similarity in composition to enable reuse of the models. Temporary tables in the database are used to store the current configuration of the model and data from

ComponentOfI and *InstanceModel* tables are used to check for similarity in composition.

- 3) The incremental instance and instance template identification number generation during the creation of the models have been clearly distinguished for the composite and atomic models.
- 4) An algorithm for identifying model names and the class files that it would correspond to in the JAVA file version of the model was developed; the generated mappings are stored in the *ModelClass* table.

3.2.2 Loading Models for simulation

This section deals with the relationship between the completion of the partially generated models, updating them with the behavior and loading them into the DEVS-Suite for simulation.

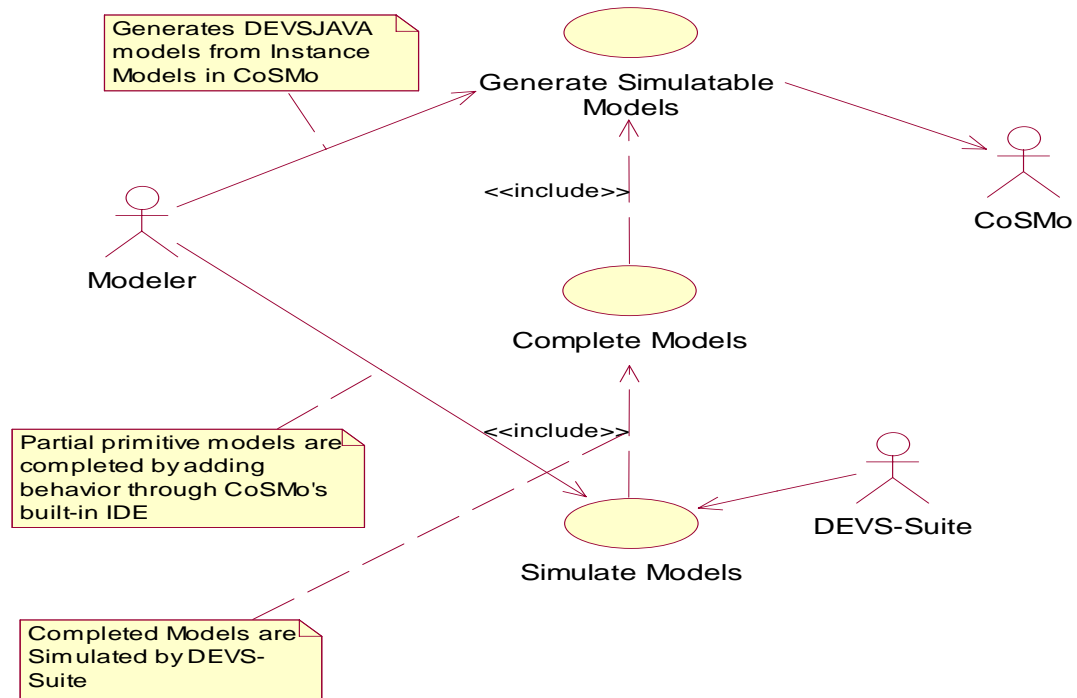


Figure 13. Loading models for simulation

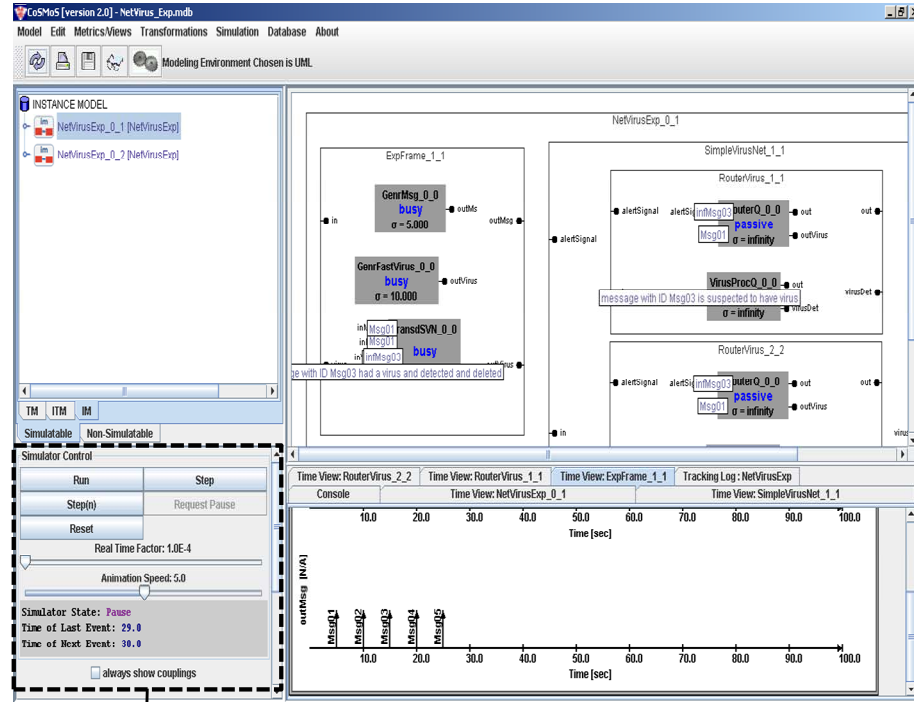
Figure 13 shows the use-case diagram outlining the important operations and interactions occurring between various actors and components in the system.

The requirements identified for the successful mapping are described below:

- 1) The model to be simulated has to be exported and saved in the workspace with a unique identifying name so there is no redundancy in the models and files.
- 2) The model selected needs to be mapped to the corresponding Java file in the predefined workspace.
- 3) The DEVS-Suite should identify the model location and the name.
- 4) The controls and the simulation view should be composed.

Approach taken to achieve these requirements for the integration is:

- 1) The VIEW of Model-View-Control of the DEVS-Suite (Figure 5) has been replaced with the CoSMo GUI.
- 2) The *ModelClass* table (Table 5) has the information of the model and the class name it corresponds to, thus the filename can be derived from it.
- 3) The exported files are arranged in a predetermined work space, which is recognized by both CoSMo and DEVS-Suite's framework.
- 4) The controls of the DEVS-Suite have been embedded into the CoSMo GUI; this helps the modeler control the simulations of a successfully loaded model.



↓ DEVS-Suite's FSimulator control

Figure 14. DEVS-Suite Simulation Controls in CoSMo

3.2.3 Visual Component and Port Selection

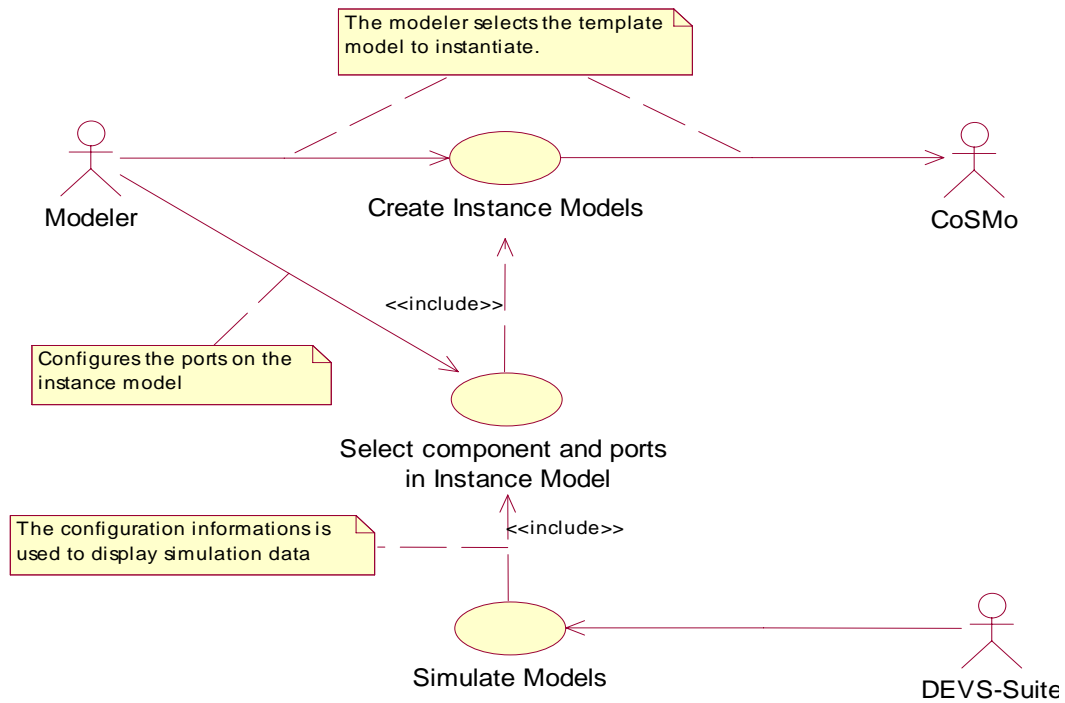


Figure 15. Visual Configuration of Models

The use case in Figure 15 describes the visual model configuration feature in CoSMo. CoSMo constantly updates its GUI with the information it receives from the mouse or keyboard events created by the modeler.

Following are the requirements identified to enable the Visual Configuration Models.

- 1) The port selected needs to be uniquely identified in the working environment.
- 2) The configuration data must be persistent.
- 3) The selection process should be reversible.
- 4) The configuration should be portable.
- 5) The configuration should be in a format that can be identified by the DEVS-Suite.

Approach taken to meet the above mentioned requirements:

- 1) A naming mechanism was used where the name of the port was concatenated along with its Model name.
- 2) The selected ports are stored in a Hash Map where the name generated in the previous step forms the Key and the port name forms the Value of the <K,V> pair in the hash map.
- 3) The port listener and the Hash Map work together to analyze the current state of the port and thus help with reversibility.
- 4) The separate input and output Hash Maps are persistent through the execution cycle.
- 5) Each entry in the Hash Map directly maps to the port trackers in DEVs-Suite; thus the information can be easily imported.

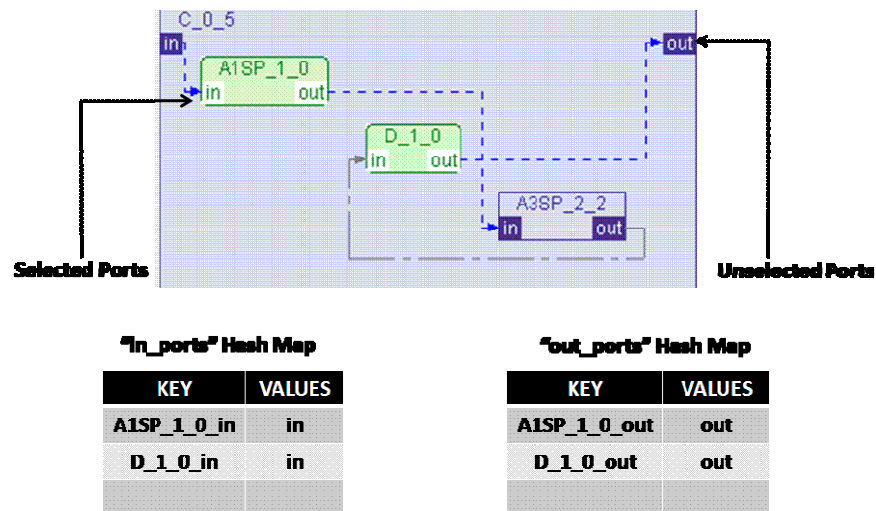


Figure 16. Visual Configuration of Models for Data Collection – The process

3.3 CoSMoS Process Lifecycle

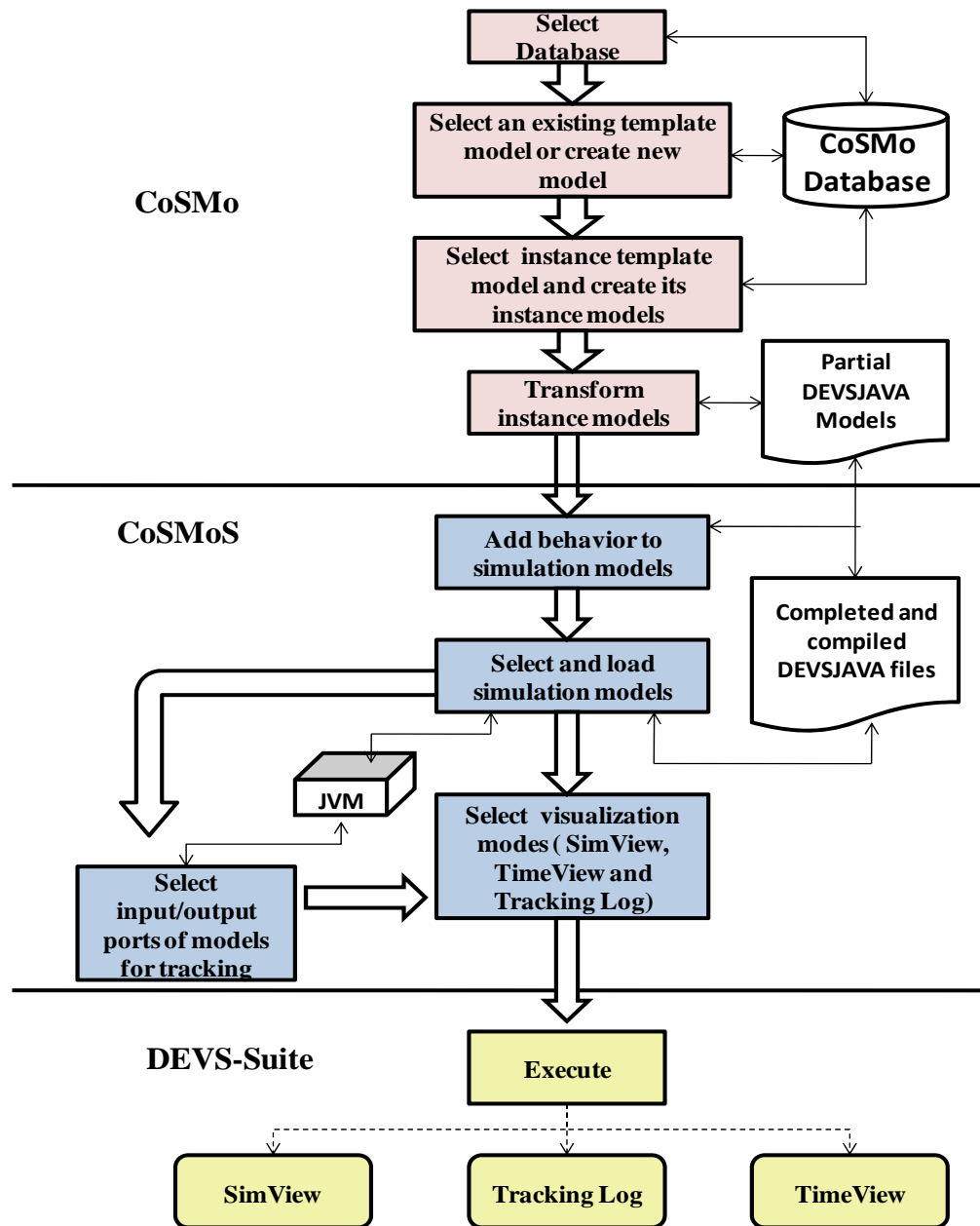


Figure 17. Process for creating and simulating models

The processes and relationships defined in Figure 17 are defined below.

Select Database: This process defines the user selecting the database that serves as a repository for the models. The relational database supports functionalities like creation, modification, storage, and reuse of the stored models. Structured Query Language (SQL) is used as a medium of communication as it is a standard language for databases and helps in application portability. The user is required to locate the database and create an appropriate data source for it using Microsoft Access (*.mdb) as the driver.

CoSMo Database: The physical database that has a predefined structure as defined by the ER schema.

Select the existing template model or create new: CoSMoS allows reuse of the models since models are stored in the database. The user can also create new, unique template models to represent a new family of models. The template model defines the primitive or composite model with input or output ports and values. The atomic model contains state variables, the ports, and the name of the model. The coupled model specifies the couplings between its components and the name of the ports. The name assigned to the primitive or the composite model must be unique, i.e., it must be identifiable within its hierarchical decomposition.

Transform Instance Models: The template models created are instantiated to a well defined model when they are transformed into Instance Models. If the model has specialized models, the user can select the specialization for these models during the transformation. The modeler can specify different models depending upon his choice

during the instantiation of template models. This gives the modeler the independence to create alternative models depending on alternative resolution and aspects.

Partial DEVJSJAVA models created: The translator in CoSMoS can export the logical models into simulation code that conforms to the syntax of the DEVJS-Suite simulation engine. The behaviors of the primitive models are defined in terms of dynamic characteristics of the model, such as input variables, output variables, state variables, and state transition functions.

Manually Add behavior to the simulation models: The primitive models are completed using the IDE in the CoSMoS environment. The models are completed by adding the behavior and completing the transition functions.

Select and load simulation models: The visual model in CoSMoS is selected to determine the model to be simulated. The models are mapped to their files that are simulation code written in JAVA. These models are complete and are compiled before the model class files are ready for simulation. It is an iterative process between **Completed and compiled Java implementation files** and **Select and load simulation models**.

Visually Select components and ports of models: The ports of the primitive and composite models can be selected visually. These selections by the user are stored in the memory (**JVM**) and are used by the *Tracking Control* in DEVJS-Suite for simulating the models.

Changes to be consistent with the models in CoSMo: The IDE in CoSMo protects the model's structure and keeps it consistent with the model specification in the database.

Select visualization modes: The modeler is given the choice of viewing the models' simulation output data on different types of trajectory viewers. The options are broadly classified into animation and tracking the simulation of the models. The animation includes the *SimView* and the tracking of the output is shown in *Tracking Log* and *TimeView*.

Execute: The complete and compiled models are simulated in the DEVS-Suite simulation engine. Depending on the selection of the visualization mode, the output trajectories are shown to the user.

Visualize: The simulation results may be viewed as time graphs (time-based trajectories, tabulated form, animation) or exported as CSV files for user-defined analysis.

3.4 Verification and Validation using CoSMoS

The CoSMoS allows a modeler to create models, generate simulation code, complete simulation code, configure models for observation, simulate models, and view simulation results. Integral to these activities are model verification and simulation validation. Since CoSMoS can be used to develop DEVS models, it can be seen that the model development and simulation process flow shown in Figure 17 supports all the FEDEP phases shown in Figure 7. The CoSMo activities can be associated with the

Develop Design and Develop Conceptual Model phases. The Implement and Tests can be associated with the CoSMoS and DEVS-Suite. The Validate Conceptual Model, Verify Design, and Verify Simulation and Validate Results are supported in part by CoSMoS's logical specification, visual model development, and model repository and DEVS-Suite's experimentation configuration and automated data generation and collection. The verification for structures of models including their interfaces and implementations is partially automated and simplified in CoSMoS. Some other aspects of model verification and simulation validation processes must be carried out manually. In particular, the completion of source code for the DEVS atomic models' functions is impractical to automate in any existing tool unless models are restricted for a particular domain (e.g., modeling and simulation of electrical circuits) and comprehensive pre-built model libraries are available. However, when model components are already verified and validated, they can be synthesized to create more complex models and benefit from the model development and simulation automation supported by the CoSMoS environment.

4 CoSMoS DESIGN AND IMPLEMENTATION

The creation of the instance models in the previous version was not controlled. Thus the modeler could create any number of instances of a given model. This led to duplicate JAVA files. The concept of a well-formed relationship between a single class with multiple instantiations (i.e., objects) could not be achieved as each model had a separate file associated with it. The logical specification for the Instance Models had to be changed to enable reuse of models with the same structures across same or different levels of hierarchy.

The persistent modeling represented by the ER schema had to be altered to reflect the new changes for the instance model creation. A table called *ExportTempTable* is defined to hold temporary values of the model components and ports to be tracked. Another table called *ModelClass* is defined to store Instance Model names and their handle names. The Instance Model names are used to create unique Java files. The handle names are those that are utilized in the Java file. The *ModelClass* table stores class-object relationships. That is, the *ModelClass* table defines the names of the classes for which unique handle names are created.

The model translator was updated to accommodate the changes in the logic of the model creation and the database structure. The structures of the created JAVA files have been defined to support their use in the DEVIS-Suite environment. This also involves the specification of new namespace for the models as the directory structure for the storing and loading of the models had been unified across the new integrated environment.

The visual modeling feature was modified to enable the user to select the ports of the models for tracking. The tracking is defined for instance models since they can be

transformed to simulation code. To select models for tracking, there must not be any side effects on the Template or Instance Template Models – i.e., mode creation, modification, or deletion is not allowed. The visual tracking of the models' ports is a single atomic process and needs to be fully completed before the models can be loaded for simulation. The information about model selection is maintained. The name of every Template Model is used to look up the corresponding class in the *ModelClass* table. After the Java models are exported and loaded into the DEVS-Suite, the simulation engine is initialized. The simulator controls are defined for the simulator and a reference to those controls loads them into the UI as shown in Figure 14. The control allows the user to execute the simulation as required and the input and output variables for the selected model components can be viewed.

4.1 *Instance Model Creation*

Once the Instance Template Models have been created, the instance creation plays an important role as it defines their realization into Instance Models. These Instance Models have direct mappings into their Java files – every Instance Model is transformed into simulation model subject to the class-object relationship defined above. Allowing the modeler to create or duplicate Instance Models (and thus simulation models) is unmanageable. To avoid this problem, the CoSMo's rules for assigning IDs to Model Instances were revised. The revised rules are described next:

- 1) TM (Template Model):

The models are classified as primitive, composite, or specialized. The composite models can be either isomorphic or homomorphic in regards to each other.

- a) tID (Template Model ID) : The tID are unique identifications, the database does not allow the duplication of tID since they form the primary key in the '*Template Model*'.
 - i) $tID \in \{0, \dots, 9\} \cup \{a, \dots, z\} \cup \{A, \dots, Z\}$; $0 < tID \leq K$; $K=54$
 - ii) The *modelType* for a component identifies if it is a primitive, composite, or specialized model.
- 2) ITM (Instance Template Models):

The Instance Template Models are added for every occurrence of a composite or primitive model. The Instance Template ID for a model specifies if it is a root component (composite) or a part component (primitive or composite).

a) Primitive Component:

- i) tiID (template instance model ID) = 0. The ITM ID of primitive models are always 0.
- ii) tID • tiID : In the Instance template model view the concatenated tID (template model ID) and tiID forms an unique representation of the model.
- iii) Concatenates tID and tiID

b) Composite Component:

- i) If model is root model
- (1)tiID =0

(2) $tiID \bullet tiID$ uniquely represents the model in the ITM view of the models.

ii) If the model is a composite component of a root model.

(1) $tiID$ is unique, $tiID \in \{1, \dots, m\}$, $m \neq \infty$.

(2) The $tiID$ in this scenario is generated based on the information in the database, i.e., if a model of the same structure exists but at a different level of hierarchy, the instance template IDs are incremented by 1.

3) IM (Instance Model):

There can be multiple instances of the same model. But the instance generated cannot be identical to each other – i.e., the configuration of the composite model based on the specialized models has to be different.

a) Primitive Component :

i) If model is root model

(1) $iID = 0$

(2) $tiID \bullet tiID \bullet iID$ uniquely represents the model in the IM view of the models

ii) If a component model, the iID , is calculated based on the information stored in the database.

b) Composite Component :

i) iID is generated based on the information present in the database. The difference is seen when the model IDs are concatenated together to uniquely identify the models.

The combination of tiID (Instance Template Model ID) and iID (Instance Model ID), if iID is 0, it shows that it's a primitive root model. The above specification was used to design and implement the algorithms described next.

4.1.1 Algorithms

We used the above rules to design the following two algorithms. Algorithm 1 prevents creating duplicate Instance Model names. Algorithm 2 defines pairs of model names and class names where each model name refers to the Instance Model name and each class name refers to the simulation model name.

4.1.1.1 Algorithm 1 – Unique Instance Model Creation

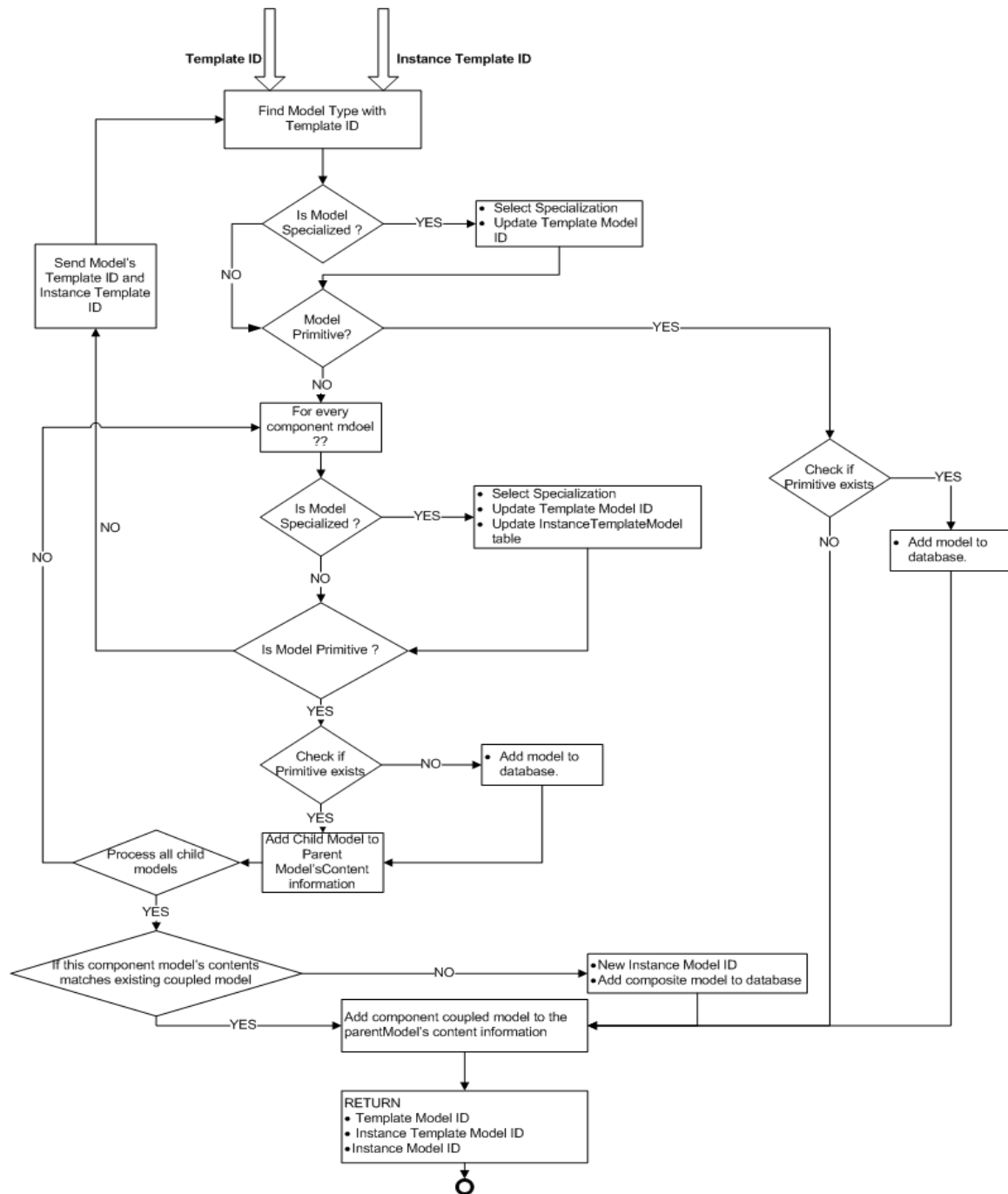


Figure 18. Unique Instance Model creation – The Algorithm

4.1.1.2 Algorithm 2 – Model Name-Class Name Relationship Creation

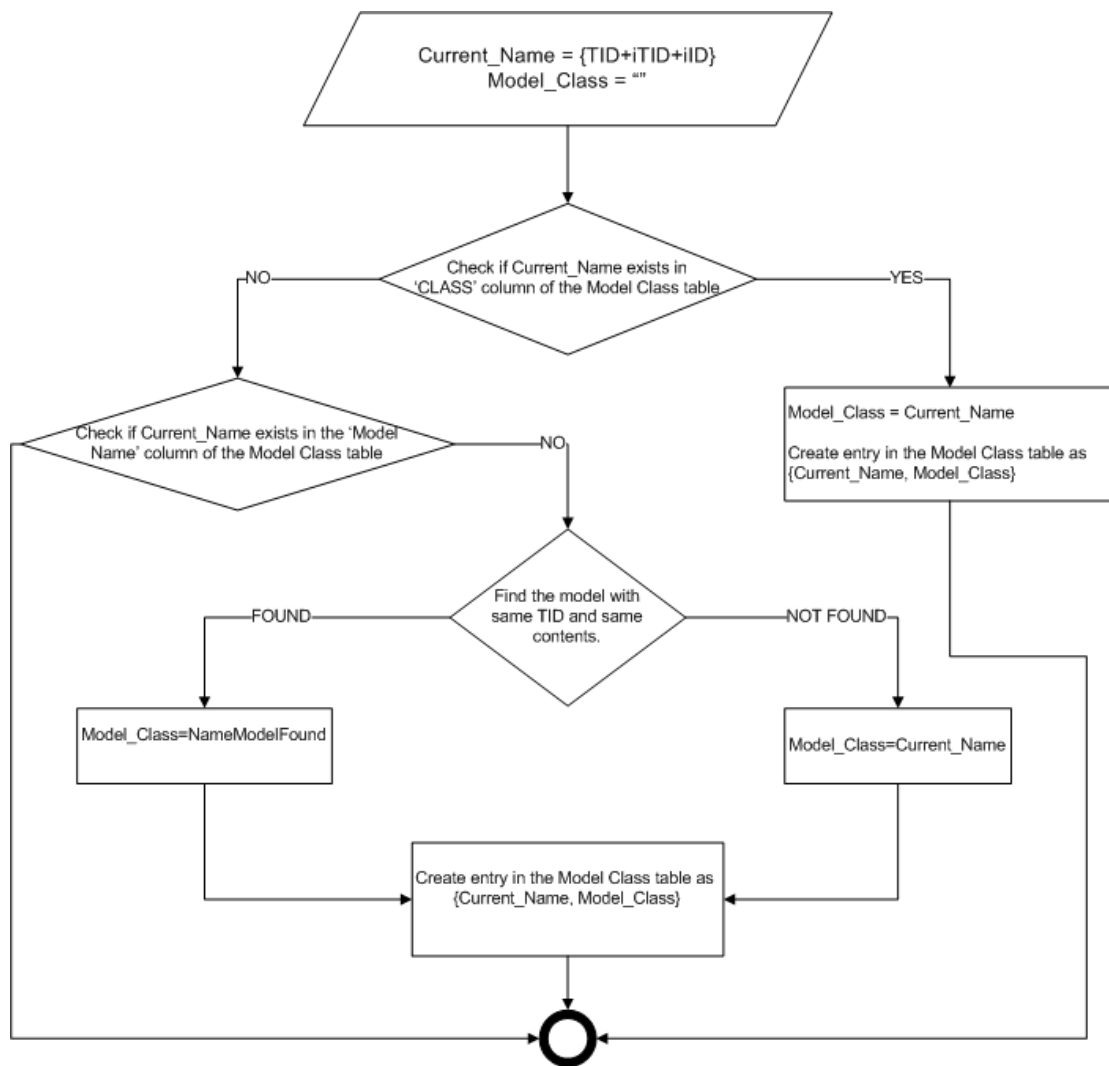


Figure 19. Model-Class Relationship – The Algorithm

4.1.2 Class Diagram

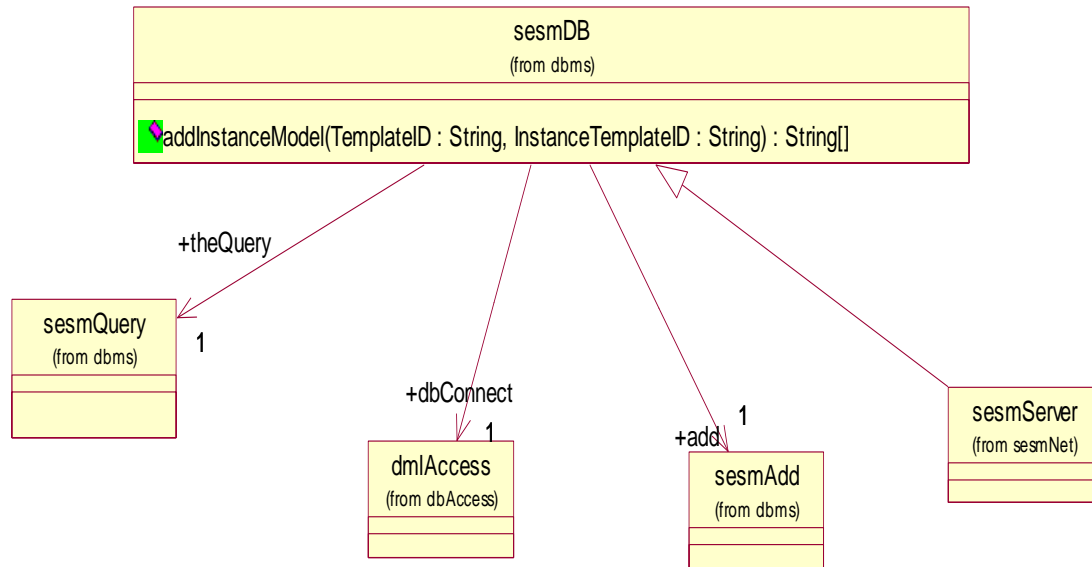


Figure 20. Class Diagram – Adding Instance Models

The *sesmDB* class has the declaration of the function that adds new instance models to the database. The function has frequent interaction with the database for making decisions regarding the creation of these instances.

Most of the interactions are in the form of queries to the database. Query operations such as building the query, query execution, and collection of the results are handled by the *sesmQuery* Class.

Once the decision has been made to create new instance models, the *sesmAdd* function performs the queries to add new instance models into the Instance Model table. The *sesmAdd* class is also used to update the *Instance Template Model (ITM)*. This is to update the specialized models ITM information in the *ITM* table.

The function for adding the *Instance Models* is called from either *sesmServer* as a user instruction or as a part of the recursive call from the same function itself. *sesmServer* inherits the *sesmDB* class and the function from the super class is called.

4.1.3 Sequence Diagram

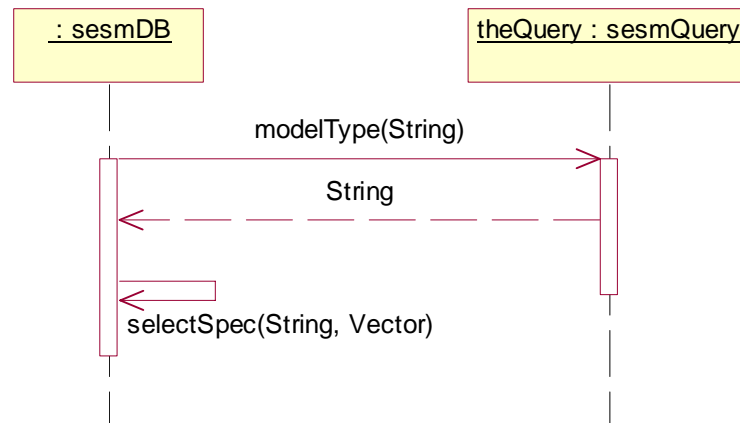


Figure 21. Sequence Diagram – Select specialization

The sequence diagram in Figure 21 shows the sequence of functions and messages passed for selecting a specialized model for the specializee model.

`modelType(String)` : This function returns the type of the model `{PRIMITIVE, COMPOSITE, SPECIALIZED}` that is identified by the Template ID which is passed as the `String` argument.

`selectSpec(String, Vector)` : This function helps the user to decide the specialization for the specializee model. First parameter refers to the Template ID of the specializee model and the second parameter stores the hierarchical model information of the model currently being specialized. On successful completion of the process, the function returns the Template ID of the specialized model.

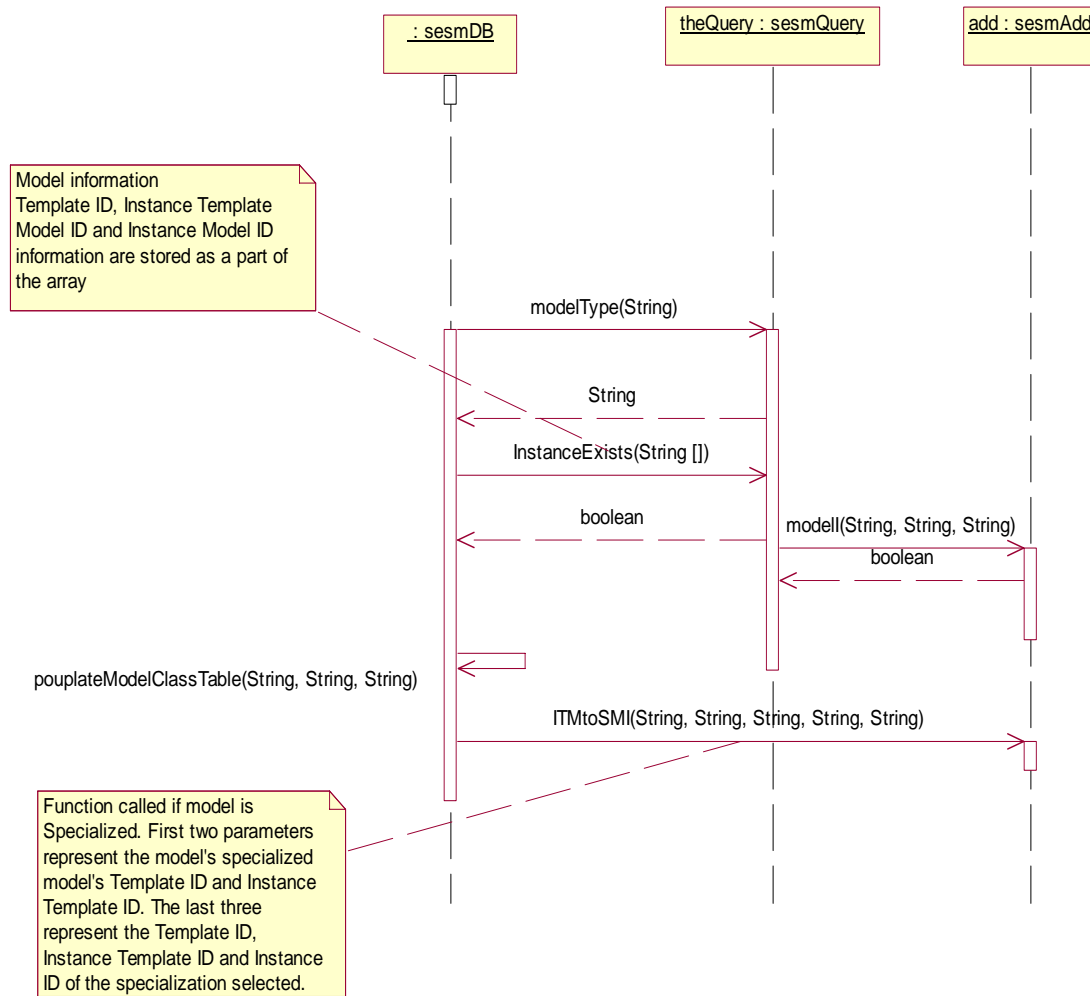


Figure 22. Sequence Diagram – Creating Instance of the Primitive Model

The Instance ID of all primitive models is '0', using the information about the Model's Template ID, Instance Template ID, and Instance ID to check if an instance of the primitive model exists in the database. Figure 22 shows the process of creating an instance model of a primitive model.

InstanceExists(String, String, String) : Checks the above mentioned and returns a Boolean value depending on the success or failure of the operation.

`modelI(String, String, String)` : This function in the *sesmAdd* class adds the instance model with the model's Template ID, Instance Template ID, and Instance ID that was generated in *sesmAdd*.

`PopulateModelClass (String, String, String)` : This function updates the *ModelClass* table with the model's details and the class name.

`ITMtoSMI(String, String, String, String, String)` : If the primitive model was a specialized model, the information about the specializee and the specialized model is stored in the *ITMtoSMI* table.

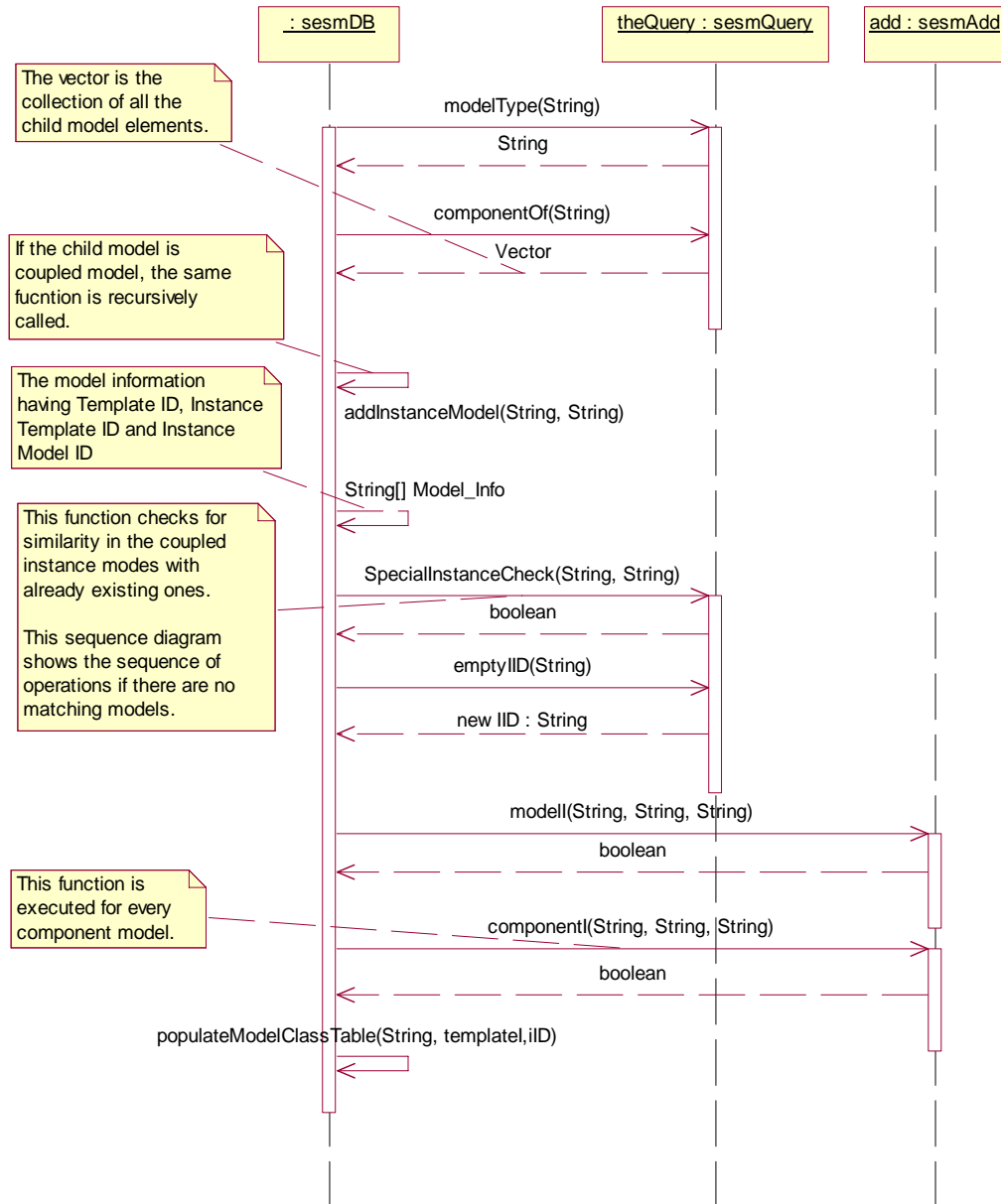


Figure 23. Sequence Diagram – Creating Instance of the Composite Models

The diagram in Figure 23 shows the sequence diagram for the generation of composite instance models. A temporary variable is used to hold the information about all of the children of the composite model. The details about the children are used to find existing models with the same constitution.

`componentOf(String)` : Returns the list of all immediate children of the model. Each of these children is processed before their parent. The `addInstanceModel(String, String, String)` function is called recursively for the children.

`SpecialInstanceCheck(String, String)` : The list of the children added to the temporary variable are run through the database to see if there exists a composite model with the same set of children.

`emptyIID(String)` : If a new instance of the model has to be created, the new instance ID is generated using the existing information in the database.

`componentI(String, String, String)` : This function updates the `componentI` table; this holds the information about the models' instances and their respective children.

These set of class diagrams, sequence diagrams, and operations define the new algorithm for adding the instance models.

4.1.4 Entity Relationship Changes

To support the new features, two new entities were added to the database.

4.1.4.1 ModelClass (*Model Class*) Entity

- Attributes
 - *template* (Template ID)
 - *templateI* (Instance Template ID)
 - *iID* (Instance ID)
 - *class* (Name of the Class)
 - *createTime* (Time of creation)

- Description

The *ModelClass* table defines the relationship between the instance models and their corresponding class. The set of IDs $\{template, templateI, iID\}$ identifies the instance model. The corresponding class names can be the same as the instance models or they can vary. The naming scheme in section 4.1 helps to establish these relationships.

The table *ModelClass* has an identifying one-to-one relationship with the *InstanceModel* table on the *template*, *templateI*, and *iID*. For every model's entry in *ModelClass* table, an exact match should exist in the *InstanceModel*. The table is updated with the Algorithm-2 shown in Figure 18.

Table 5

Relational Database Schema Specification for ModelClass Table

Model Class				
<u>template</u>	<u>templateI</u>	<u>iID</u>	class	createTime

As seen in Table 5, $\{template, templateI, iID\}$ forms the primary key from the table.

- *template* is a foreign key from InstanceModel (*template*).
- *templateI* is a foreign key from InstanceModel (*templateI*).
- *iID* is a foreign key from InstanceModel (*iID*).
- *class* is an alphanumeric string with a maximum length of 50 characters.
- *createTime* is a double decimal number.

4.1.4.2 Extended CoSMoS Transactions

Addition of the new algorithms requires the addition of a temporary disjoint table, *ExportTempTable*.

Table 6

Relational Database Schema Specification for ExportTempTable Table

ExportTempTable					
<u>tComponent</u>	<u>tiComponent</u>	<u>iComponent</u>	tOwner	tiOwner	iOwner

The relationship between the newly table and the existing schema is show in Figure 24. The *ModelClass* table forms an identifying relationship with the *InstanceModel* table. Thus every record in the *ModelClass* table can be identified by an entry in the *InstanceModel* table. This table is updated by the algorithm specified in Figure 19. The *ExportTempTable* forms an identifying relationship with the *componentOfI* table. This table and the relationship help in generating the query that would check for identical composition of models in the database. The identifying relationships between the models helps in the cascade add and deletion operations.

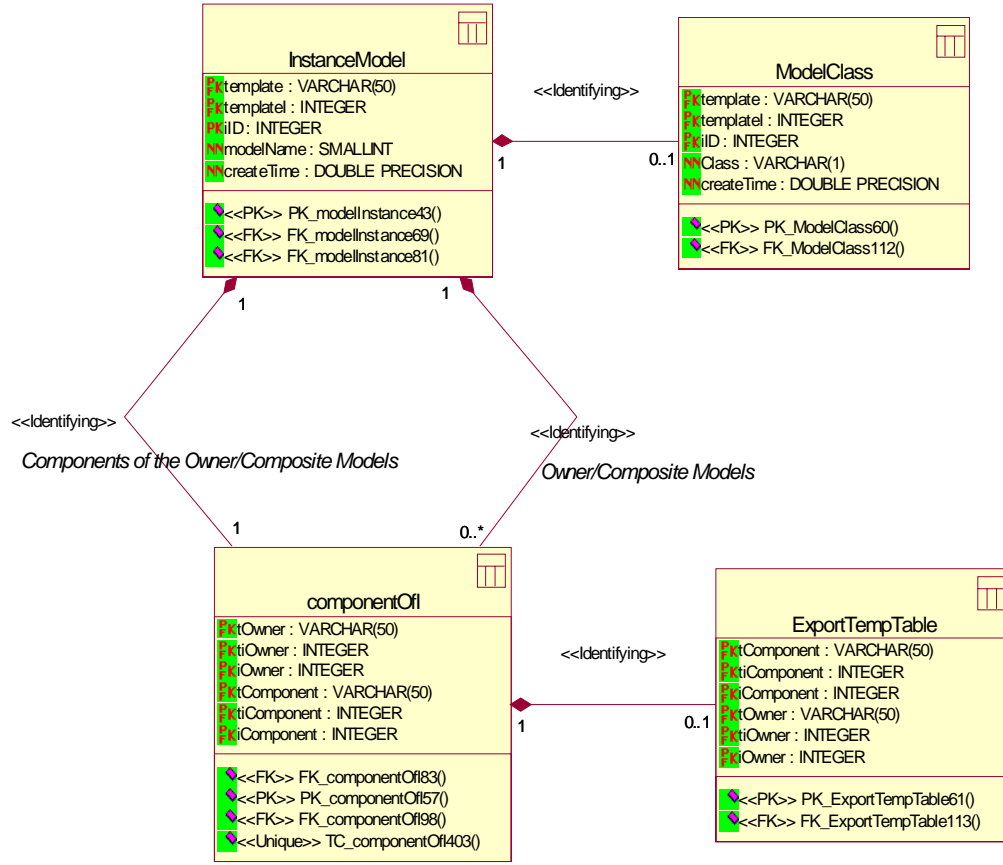


Figure 24. Entity Relationship Changes

The SQL query for a transaction to check the similarity of a model being created in terms of its composition with already existing models is given below. There are two tables needed for identifying the similar models *ExportTempTable* and *InstanceModel*.

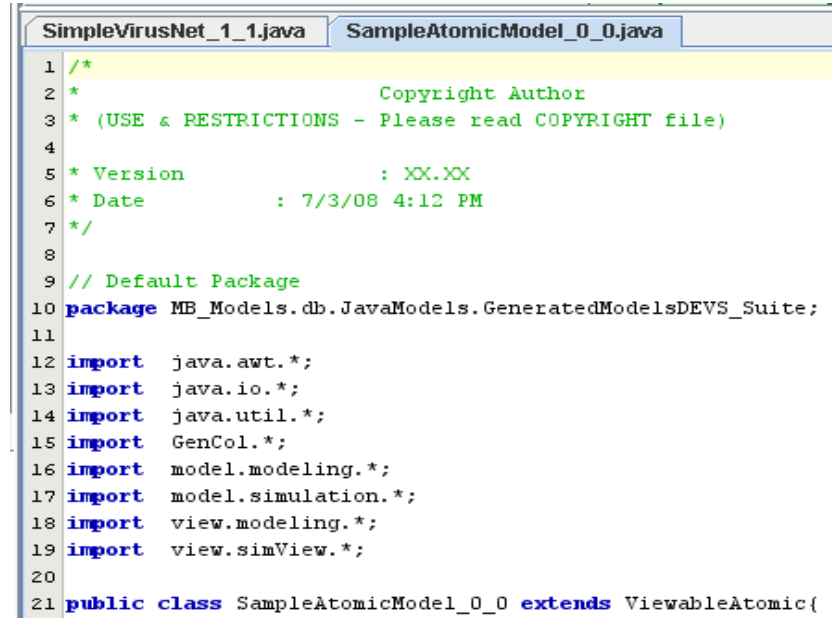
$$\begin{aligned}
 & \pi_{tOwner, tiOwner, iOwner, tComponent, tiComponent, iComponent} (componentOfI) \\
 & \div \\
 & \pi_{tComponent, tiComponent, iComponent} \left(\sigma_{(tOwner='ExpFrame' \text{ AND } tiOwner=0)} (ExportTempTable) \right)
 \end{aligned}$$

- `SELECT DISTINCT tOwner, tiOwner, iOwner FROM (SELECT tOwner, tiOwner, iOwner, tComponent, tiComponent, iComponent FROM componentOfI) T WHERE tiOwner = 0 AND not exists (SELECT * FROM (SELECT DISTINCT tComponent, tiComponent, iComponent FROM ExportTempTable WHERE tOwner = 'ExpFrame' and tiOwner=0)B WHERE NOT EXISTS (SELECT * FROM (SELECT tOwner, tiOwner, iOwner, tComponent, tiComponent, iComponent FROM componentOfI)AB WHERE ((AB.tOwner = T.tOwner AND AB.tiOwner = T.tiOwner AND AB.iOwner = T.iOwner) AND (AB.tComponent = B.tComponent AND AB.tiComponent = B.tiComponent AND AB.iComponent = B.iComponent)))))`

4.2 *Export Models*

4.2.1 *Exported Models File Structure*

The primitive and composite models from CoSMoS's visual models can be transformed to two types of simulation models. For DEVS-Suite, CoSMoS generates atomic and coupled simulation models. The created Java files conform to the DEVS specification. These simulation models have their own unique structures.



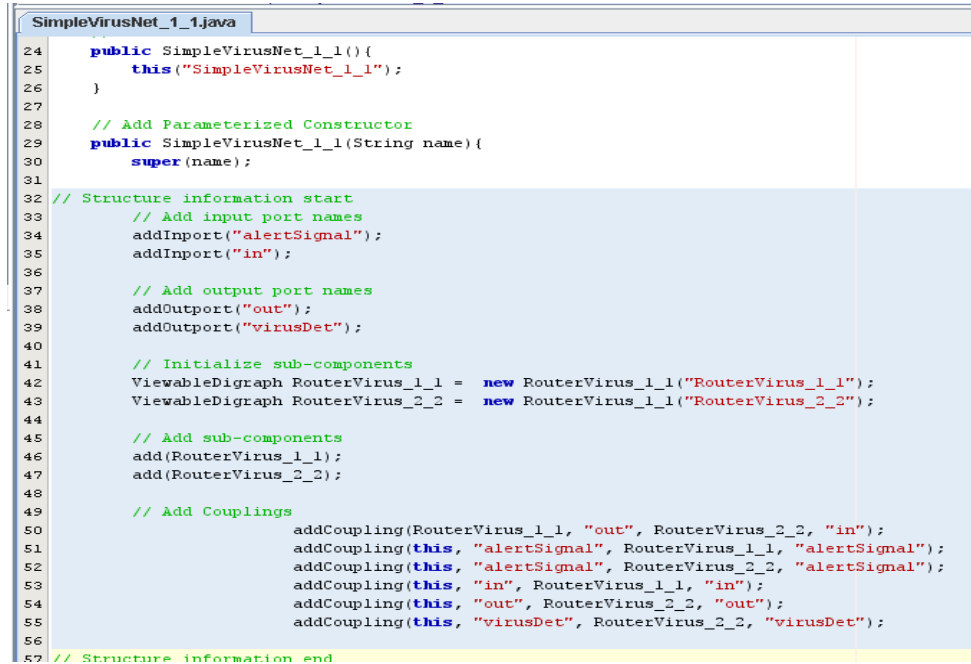
```

1  /*
2  *                               Copyright Author
3  * (USE & RESTRICTIONS - Please read COPYRIGHT file)
4
5  * Version                       : XX.XX
6  * Date                         : 7/3/08 4:12 PM
7  */
8
9  // Default Package
10 package MB_Models.db.JavaModels.GeneratedModelsDEVS_Suite;
11
12 import java.awt.*;
13 import java.io.*;
14 import java.util.*;
15 import GenCol.*;
16 import model.modeling.*;
17 import model.simulation.*;
18 import view.modeling.*;
19 import view.simView.*;
20
21 public class SampleAtomicModel_0_0 extends ViewableAtomic{

```

Figure 25. A sample of a generated atomic model

Figure 25 shows a partial sample atomic model created by the model transformation from CoSMoS. Changes were also made to the coupled model generation. The model transformation function becomes more with the atomic model as it has to establish the object-class relationship using the *ModelClass* table. The new instance model creation algorithm that has been added to CoSMoS populates the coupled model with the instances of the models that belong to it. Every (atomic or coupled) model that is part of the coupled model is instantiated from a class and has a handle name. When adding in the components for the coupled model, every model is checked for the corresponding class in the *ModelClass* table. A snippet of a coupled model is shown in Figure 26.



```

SimpleVirusNet_1_1.java
24 public SimpleVirusNet_1_1(){
25     this("SimpleVirusNet_1_1");
26 }
27
28 // Add Parameterized Constructor
29 public SimpleVirusNet_1_1(String name){
30     super(name);
31 }
32 // Structure information start
33 // Add input port names
34 addInputPort("alertSignal");
35 addInputPort("in");
36
37 // Add output port names
38 addOutputPort("out");
39 addOutputPort("virusDet");
40
41 // Initialize sub-components
42 ViewableDigraph RouterVirus_1_1 = new RouterVirus_1_1("RouterVirus_1_1");
43 ViewableDigraph RouterVirus_2_2 = new RouterVirus_1_1("RouterVirus_2_2");
44
45 // Add sub-components
46 add(RouterVirus_1_1);
47 add(RouterVirus_2_2);
48
49 // Add Couplings
50 addCoupling(RouterVirus_1_1, "out", RouterVirus_2_2, "in");
51 addCoupling(this, "alertSignal", RouterVirus_1_1, "alertSignal");
52 addCoupling(this, "alertSignal", RouterVirus_2_2, "alertSignal");
53 addCoupling(this, "in", RouterVirus_1_1, "in");
54 addCoupling(this, "out", RouterVirus_2_2, "out");
55 addCoupling(this, "virusDet", RouterVirus_2_2, "virusDet");
56
57 // Structure information end

```

Figure 26. A sample of a generated coupled model

4.2.2 Model Namespace

The namespace for the models that are converted into simulation code has been changed to reflect the new implemented directory changes. The location for all the generated simulation code is relative to the working directory. The location of these JAVA files are based on the database selected. Separate folders are maintained for each database loaded. The files are arranged in a master folder at the same level of the file system where the environment main file resides. Each database utilized by the user creates a separate folder and holds the actual database file followed by the Java, XML, and NSM (Non Simulatable models) folders to hold their respective files. Figure 27 shows the layout of the folder structure for maintaining the namespace of the generated simulation models.

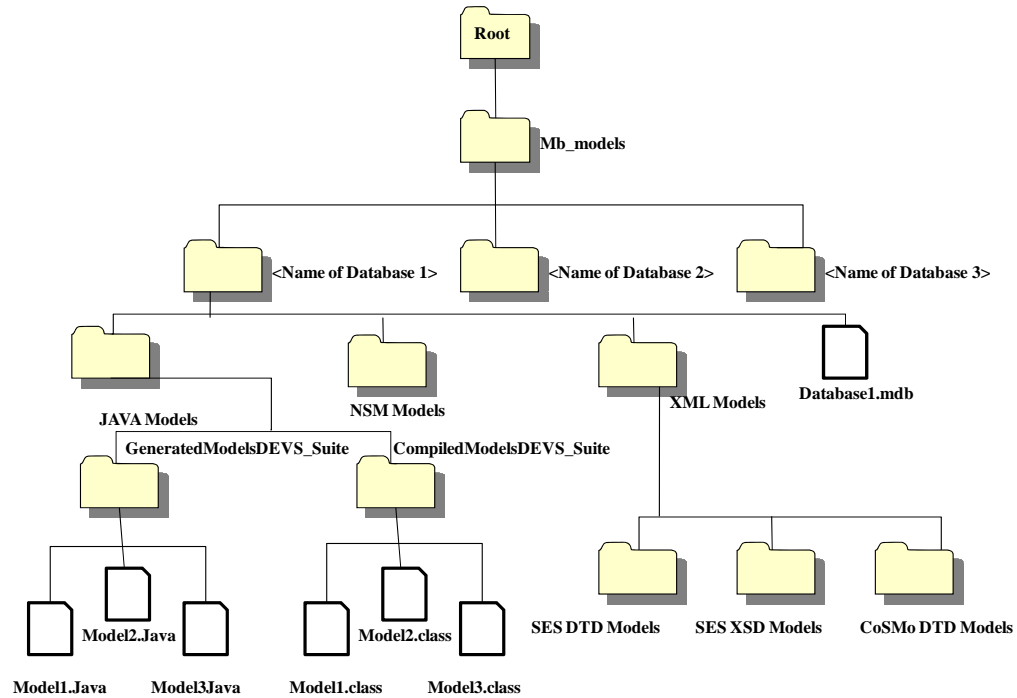


Figure 27. File-Directory Structure

4.2.3 CoSMo Editor

The atomic models that are transformed from CoSMoS are to be completed before they can be loaded into the simulator and simulated. CoSMoS assists the user in editing the JAVA file and adding behavior in it. The behavior is added to the model in terms of internal transition, external transition, confluent, and output functions as well as model initialization. The structural information in these Java model files are automatically added during the transformation according to the database. The Java file of the model has to be consistent in terms of the name, ports, variables, and state variables with the model in the database.

Figure 51 shows sample tabs of various source code editors opened up in CoSMoS. The editor is available as a part of the Netbeans editor API. The editor has functionality

such as 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. Some markers are included in the generated JAVA files that act as tags and does not allow the models to be edited. The guarded sections of the code, seen as a shaded section, can be observed in Figure 49.

4.3 Visual Model Component and Port Selection

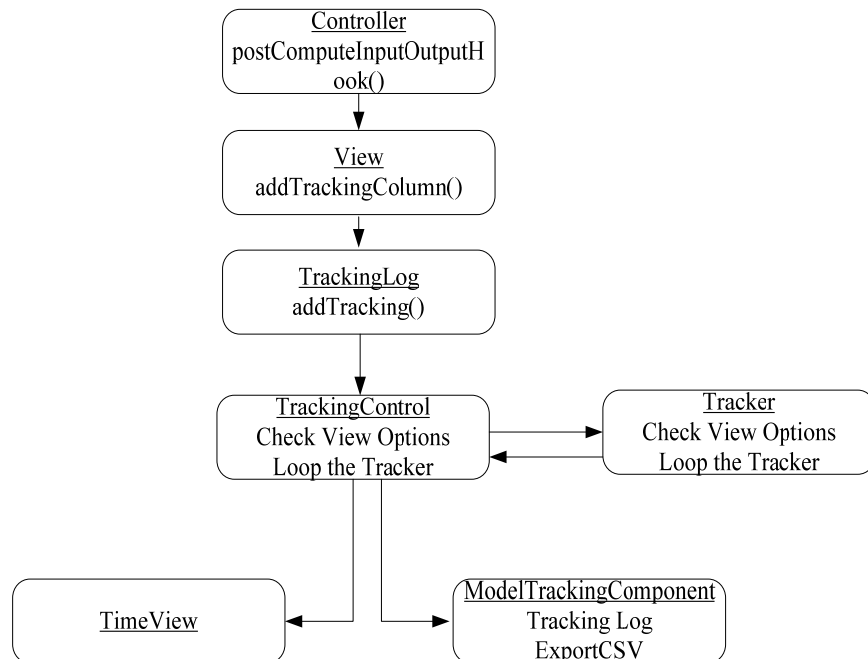


Figure 28. Data Flow in DEVS-Suite

The models loaded in the DEVS-Suite are assigned default trackers. The DEVS-Suite allows the user to select the components of a coupled model for visualizing its input and output ports as well as all of its parts. For atomic models, state variables (Phase and Sigma) and time parameters (tN: Time of next event and tL: Time of last event) can also be visualized with the help of trackers. Figure 28 shows the data flow for the DEVS-Suite.

The Controller is responsible for the creation of the hooks with the View. The View delegates the logic for determining the data for output trajectory viewers through the *TrackingControl* class. Each tracker associated with the model has Boolean checkers to enable or disable the components for tracking. These trackers can be invoked in the DEVS-Suite with the help of the tracking dialog box associated with each of the models. The user may also choose one or more model components with the ability to select one or more of its ports. Figure 16 shows the design of the feature that creates the bridge between CoSMo's visual selection and the trackers of the models in DEVS-Suite.

4.3.1 Class diagram

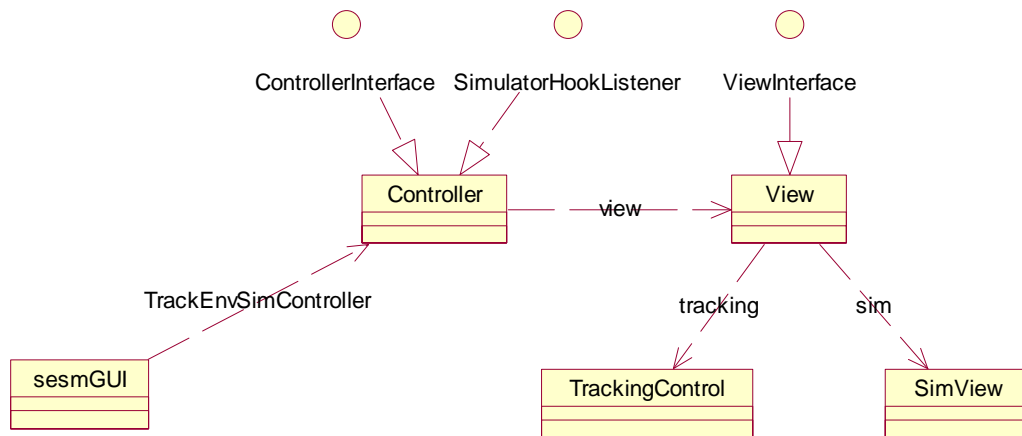


Figure 29. Class Diagram – Visually selecting components

As the Instance Models are selected from the CoSMoS for simulation, the completed simulation models are loaded from the respective JAVA files. These JAVA files are loaded into the *Controller* of the DEVS-Suite. The *Controller* invokes the *View* to initialize and setup the output trajectory viewer such as the *TimeView*, tracking log, and *SimView*. Although the *SimView* forms a part of the graphical user interface, the

TrackingControl does not control it. The *TrackingControl* forms the intermediate component for providing the output data to be displayed in the trajectory viewer.

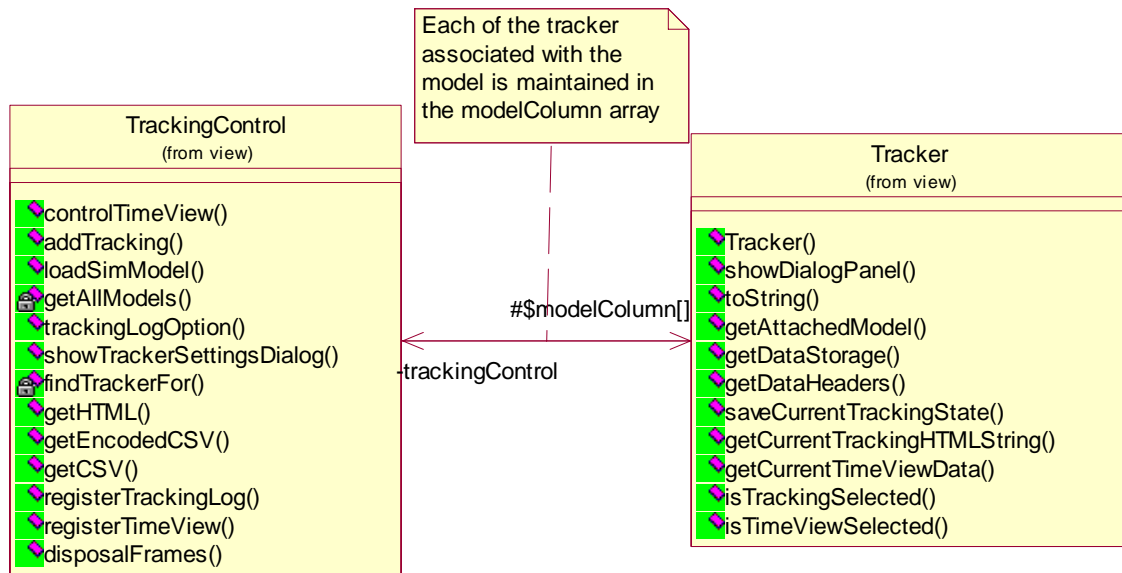


Figure 30. Class Diagram – *TrackingControl* and *Tracker*

After these trackers have been initialized as a part of the model loading process, they have to be updated based on the selection made by the user on the visual models. The scheme used for this updating process is defined in the sequence diagrams given below.

4.3.2 Sequence Diagram

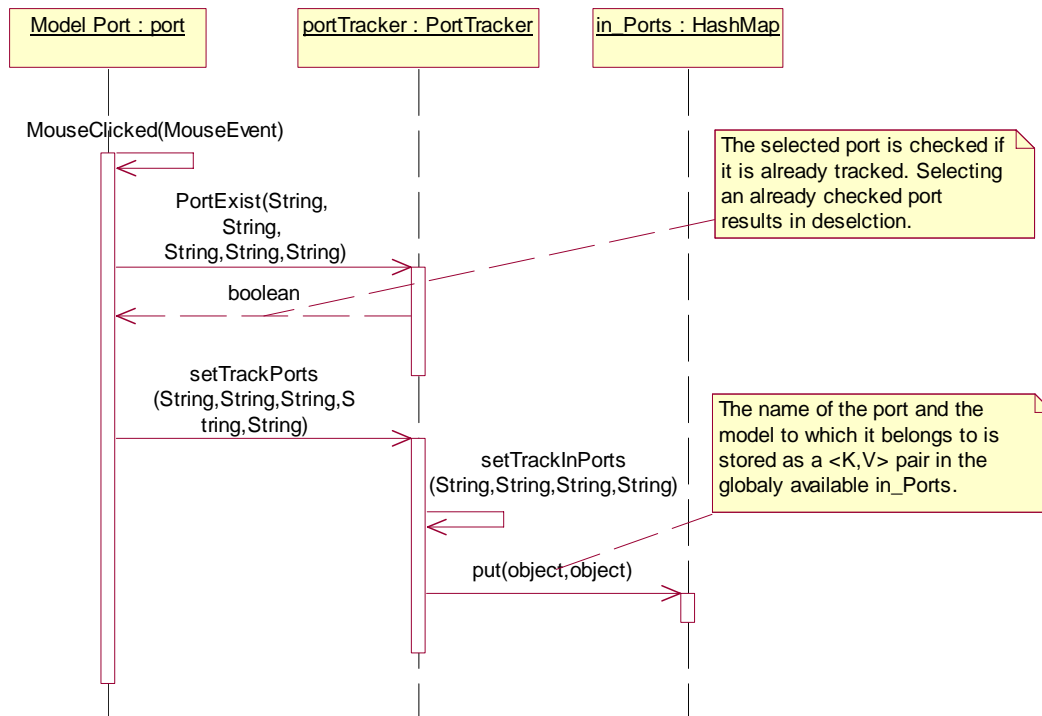


Figure 31. Sequence Diagram – Selecting input ports for tracking by mouse clicks

The ports selected by the user in CoSMoS' visual models are recorded in temporary Hash Maps. Figure 16 shows the structure of the temporary Hash Map. The sequence diagram in Figure 31 illustrates the process of selecting the ports in the models with mouse clicks. The colors of the ports and the text are reversed to represent the status of the selection. The selected ports are inserted into the temporary Hash Maps with the "Name of the model + Name of the port" as the key and the "Name of the port" as the value. The selection has the ability to toggle between tracked and ports not tracked. These selected ports make visualizing time-based trajectories possible with DEVs-Suite.

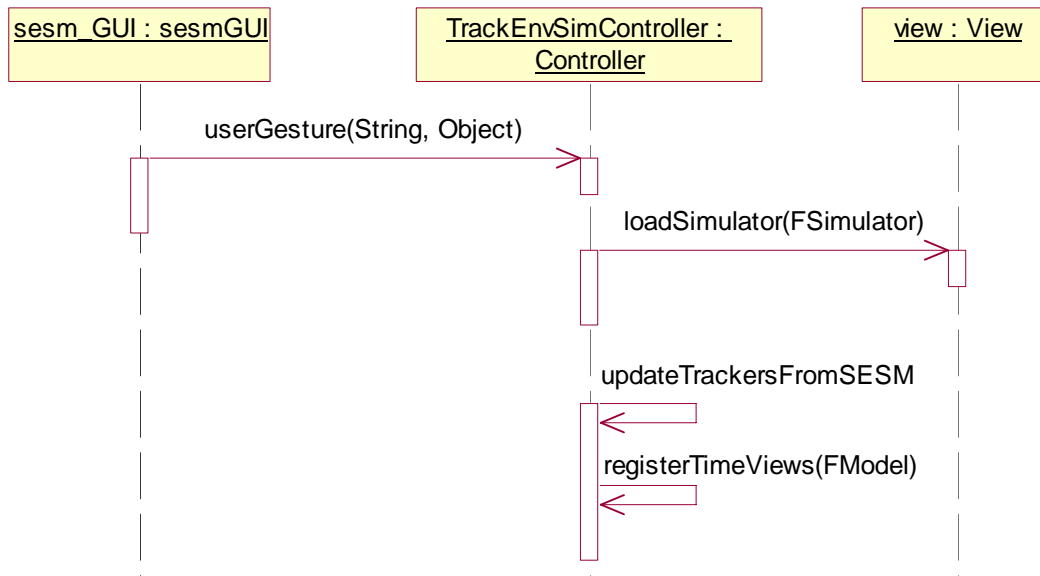


Figure 32. Sequence Diagram – Loading Models for Simulation

The process of selecting the ports for tracking has to be completed before the models can be loaded for simulation. As shown in the sequence diagram in Figure 32, the function *updateTrackersFromSEM()* from the controller sets the port trackers from DEVSSuite based on the information present in the Hash Maps from CoSMoS. When the user selects the TimeView option, the time graphs must be initialized before they can be used for adding events. The *registerTimeViews(FModel)* function in *Controller* is responsible for setting up the TimeView.

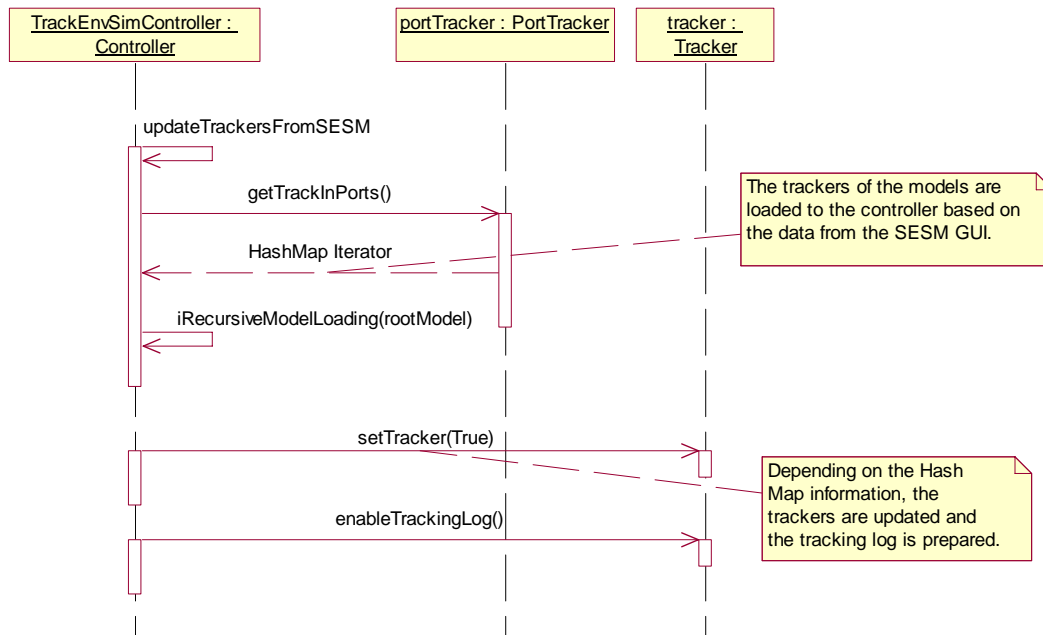


Figure 33. Sequence Diagram – Enabling models for tracking

The sequence diagram in Figure 33 explains the *updateTrackersFromSESM()* function described in Figure 32. The *controller* retrieves the Hash Maps that have the information about the user's selection from the *PortTracker* class. The sequence diagram described in Figure 33 defines the scenario involving only the input ports. The same logic applies for the output ports. The *controller* then calls the *iRecursiveModelLoading(rootModel)* where all the models in the hierarchy of the specified root model are retrieved. These models' trackers are accessed and modified based on the information extracted from the Hash maps from CoSMo. The function *setTracker (True)* is used to enable the tracking flag of a particular component of the model. The *enableTrackingLog()* is called by the controller so that the output data can be written in an HTML format in the tracking log.

4.4 Loading Models for Simulation

After the modeler has decided to simulate a completed instance model from CoSMo, the JAVA implementation model is found in the repository location specified by the property file. As already explained in Section 4.1, the creation of the instance models are changed to create unique models and eliminate redundancy. The visual models shown have classes (i.e., Java files) and may also be handles to an already existing model having exactly the same structure. The DEVS-Suite loads the Java files and generates data for *SimView* and *Tracking Control*. The visualization of the output trajectory for *SimView* and *Tracking Control* (*TimeView* and *Tracking Log*) requires different logic. If the user selects the option of *SimView*, the completed and compiled Java files are passed as parameters for the *SimView*. During simulation mode, the View pane replaces the graphics of the model in CoSMo's GUI. If the tracking option is selected by the user, the primitive and composite block models are shown and the functionality of the *Trackers* attached to the models helps in monitoring the models throughout the simulation.

4.4.1 Class Diagram

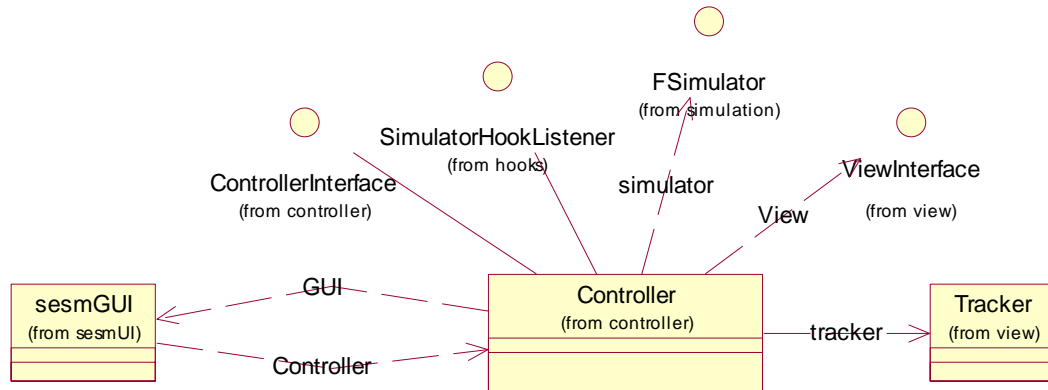


Figure 34. Class Diagram – Components for loading models

As shown in Figure 34 the class diagram delineates the classes required for loading the models into the simulator. The user gestures for the DEVs-Suite are emulated from CoSMo by specifying the location of the models. The models are loaded in the form of Java files and compiled automatically. The loading of the class files is terminated if any of the Java implementation fails to compile. Any errors are displayed in a console. The user can edit the Java files using the editor provided in CoSMoS and debug the programming errors manually. The *sesmGUI* object also retrieves the simulator and the *TrackingControl* instance setup for the model.

4.4.2 Sequence Diagram

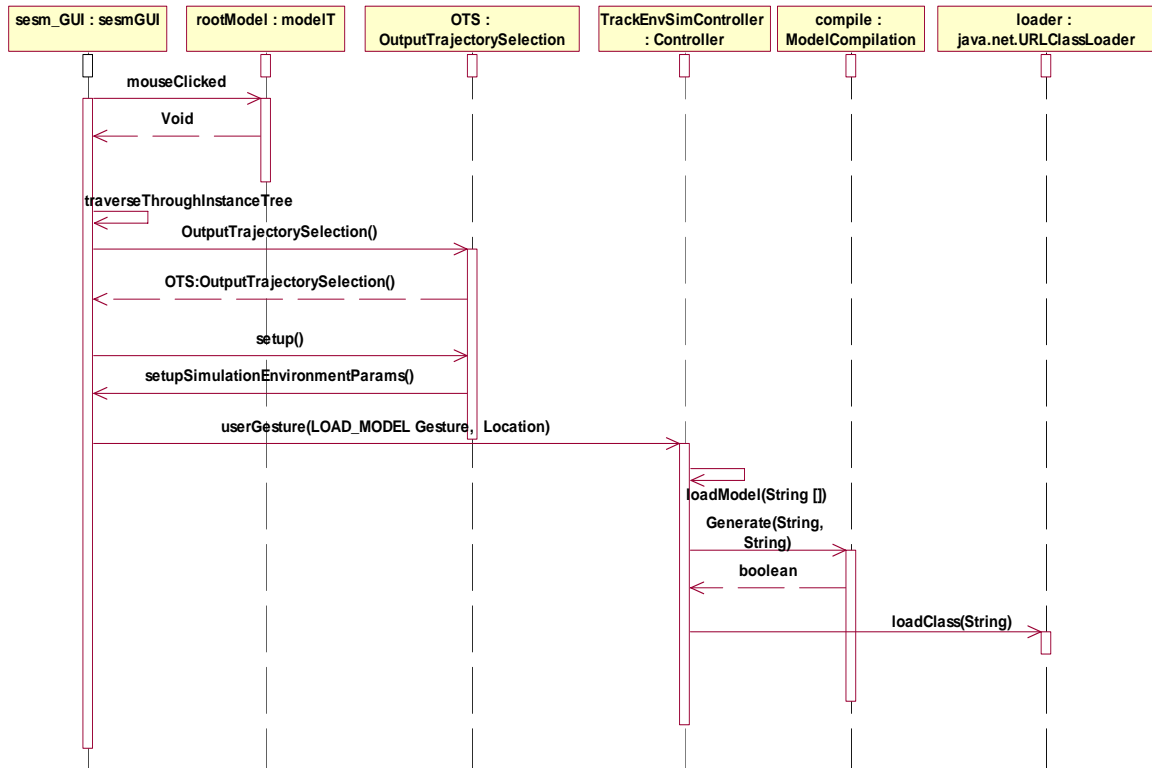


Figure 35. Sequence Diagram – Loading models for simulation

The sequence diagram in Figure 35 shows the control and the data flow between the various components needed to successfully load the models in the simulator. Once the start simulation option is chosen, the selected model is mapped to the *ModelClass* table and the classes corresponding to it are found. Using the class information, the models are loaded into the simulator.

The visual models in CoSMo have mouse listeners that register these models as the last model is selected. After the menu item *Track* has been chosen, the last selected gets registered as the root model. The user cannot select a model outside the root model's hierarchy while in the tracking mode.

A window showing different output trajectory viewers options is presented to the user. After the components to be tracked have been selected, the *setupSimulationEnvironmentParams()* is executed by the *OutputTrajectorySelection* object of the environment. The user may chose to select the SimView to view the animation of the models and the messages between them during the simulation instead of selecting models to be tracked. The SimView is loaded in the visual block model view area. After the successful completion of the setup of the Tracking environment, the command to load the model along with the model's location as arguments is executed. The user gestures for the loading of the models are performed by calling the *userGesture()* in the *Controller*. These raw Java files are compiled at first using the *ModelCompilation* class DEVS-Suite (Kim, 2008; Kim, et al., in preparation). Subsequently, the class files for the created Java files are loaded using the dynamic class loader functions available in the *java.net.URLClassLoader*.

5 DEMONSTRATION

The capabilities of the integrated environment are demonstrated with the help of a simple anti-virus network model (S. Bendre, 2004). The demonstration shows the basic life cycle of a model that involves creation of the models, instantiating, adding of behavior for simulation, configuration for simulation experiments, simulation, and viewing the simulation data.

5.1 Anti –Virus Model Example

The anti-virus model describes an anti-virus system that is intended to protect a network of computers from virus attacks. The *SimpleVirusNet* consists of two *RouterVirus* coupled models. 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.

The *RouterQ* acts as the processor for the *RouterVirus* model. If it receives a message and is not affected by a virus, the message is sent to the *out* port after processing. The type of messages arriving at the *alertSignal* port is 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.

ExpFrame, an experimental frame, was designed for the simulation experiments. The experimental frame consists of a message generator *GenrMsg* and a virus generator *GenrVirus*. The *GenrMsg* generates messages for the *in* port of the *SimpleVirusNet* and *GenrVirus* generates messages for the *alertSignal* port.

5.1.1 Select Database or Create New Database

When starting up the environment the user is given the option of selecting an existing database or creating a new database for creating and modifying the models. The first process in Figure 17 shows the selection of database and establishing the connection for the further processes.

The screenshot shows a window titled "CoSMoS MetaData Form". It contains the following fields and controls:

- User Name:** A text box containing "SesmUser".
- Password:** A text box with masked characters (dots).
- Select Mode:** Two radio buttons labeled "UML" and "XML".
- Enter Database Name:** A text box containing "ModelsDatabase" and a browse button (three dots) to its right.
- Java Files Location:** A text box containing "ase\JavaModels\GeneratedModelsDTE\".
- DTD Files Location:** A text box containing "atabase\XML Models\SESM DTD Models\".
- XSD Files Location:** A text box containing "atabase\XML Models\SES XSD Models\".
- Buttons:** "Yes" and "No" buttons at the bottom center.

Figure 36. Selecting existing database

As seen in Figure 36, the model can be loaded from an existing database. To see the list of all the existing databases in the working directory, click the button labeled with the "...". next to the "Enter Database Name:"; the drop down menu lists of all the existing databases.

The environment also empowers the user to create new database by simply adding the name of the desired database in the "Enter Database Name" text field. A separate folder with the name of the database is created in the file system.

5.1.2 Model Creation

The next step, according to the process flow in Figure 17, is shown by the block of *Selecting or Adding Template Models* in the *CoSMo* section. The model creation first involves creating the template models.

The template models can be created in two ways

- 1) Using the *Model* menu and selecting the option *Create Model Template* (Figure 37).
- 2) Using the keyboard shortcut ALT+T.

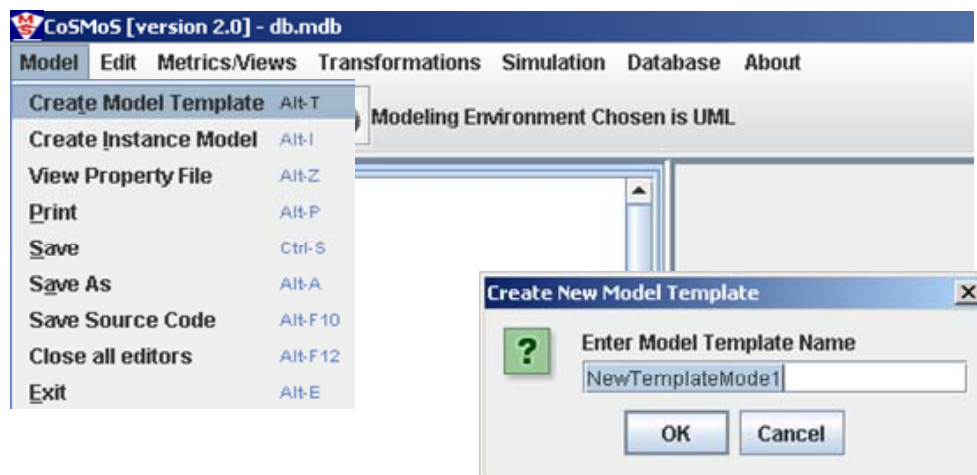


Figure 37. Creating a new Model Template

When primitive models like GenrMsg are being added, the above mentioned methods can be used to add them.

Table 7

Primitive models in the Anti-Virus Model

PRIMITIVE MODELS	DESCRIPTION	PORTS	
		INPUT	OUTPUT
GenrMsg	Responsible for generating messages and sending it to the TransdSVN and SimpleVirusNet	-	outMsg
GenrVirus	Generates the alert messages for the corresponding infected messages	-	outVirus
TransdSVN	Collects the information about the experiment.	inMsg inNet inVirus virusNet	-
RouterQ	Processes the messages generated by the generators.	alertSignal in	out outVirus
VirusProcQ	Checks if the message is infected with the virus.	inVirus	out virusDet

The rest of the models are initially created as primitive models and then components are added which convert them into composite models. During the process of adding components to a composite model, a dialog box pops up that accepts the multiplicity for the component in that model.

Table 8

Composite models in the Anti-Virus Model

COMPOSITE MODELS	COMPONENTS (MULTIPLICITY)	PORTS	
		INPUT	OUTPUT
ExpFrame	GenrMsg(1) GenrVirus(1) TransdSVN(1)	in virus	outMsg outVirus
RouterVirus	RouterQ(1) VirusProcQ(1)	alertSignal in	out outVirus
SimpleVirusNet	RouterVirus(2)	alertSignal in	out virusDet
NetVirusExp	SimpleVirusNet	-	out virusDet

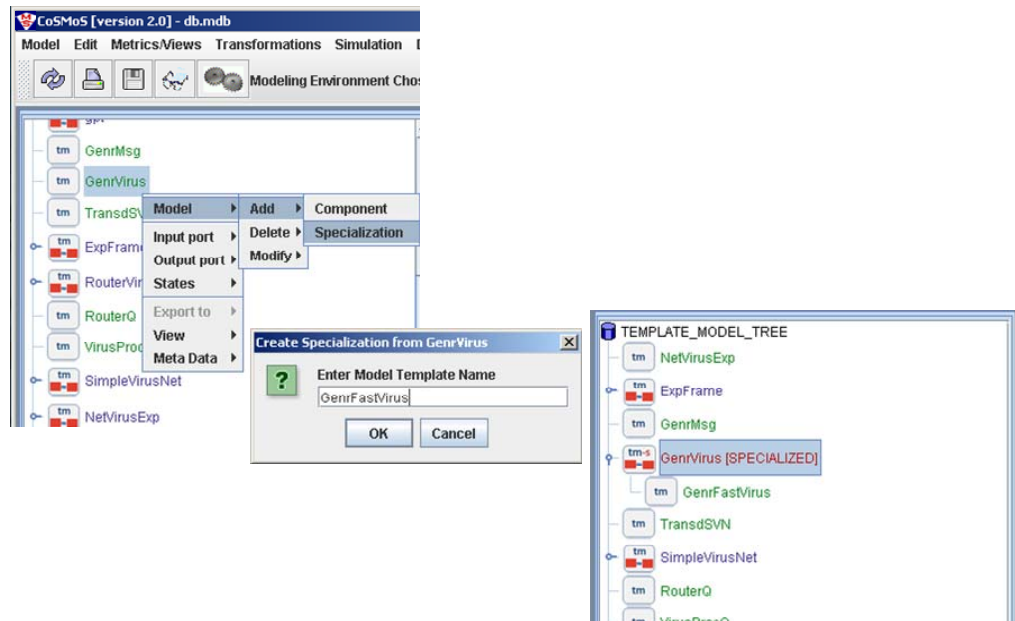


Figure 38. Adding specializations to a model

The template model can be specialized into one or more specialized components. Specialization supports different types of models to be instantiated based on the intent of

the modeler. The model can be specialized using the *is-a* relationship. Figure 36 shows the process of adding specialization to a model using the popup menus on the tree.

The specializations can be added in the following ways:

- 1) Select the component on the tree and right click on the node which brings up the menu associated with the model. The selection of *Model* → *Add* → *Specialization* helps in adding the specialization for the model (Figure 38).
- 2) The menu can also be invoked by a right click on the model in the model display panel.

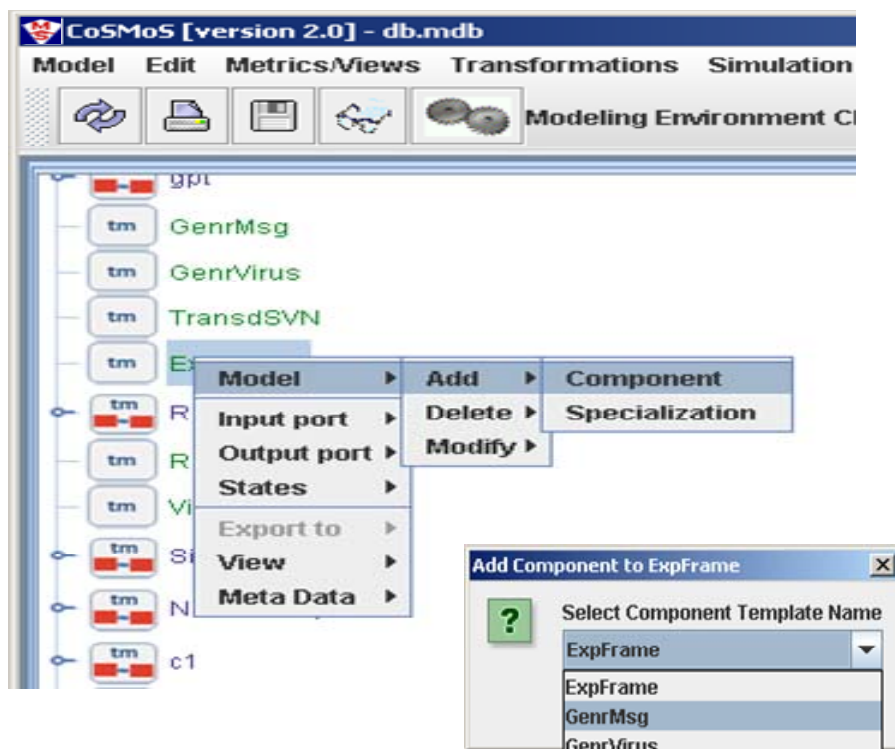


Figure 39. Adding components to a model

The add component command can be called by the following ways:

- 1) Select the component on the tree and right click on the node to bring up the popup menu. The selection of *Menu → Add → Component* helps in adding the component. The process is shown in Figure 39.
- 2) The same process can be carried out at the model display panel; a right click on a model brings up the same menu described for the tree view.
- 3) The option is also available in the Menu bar under the *Edit → Add Component*. The model to be operated upon should be under focus in the tree view.
- 4) The command mentioned in option 3 can also be accessed by using the keyboard shortcut ALT+C.

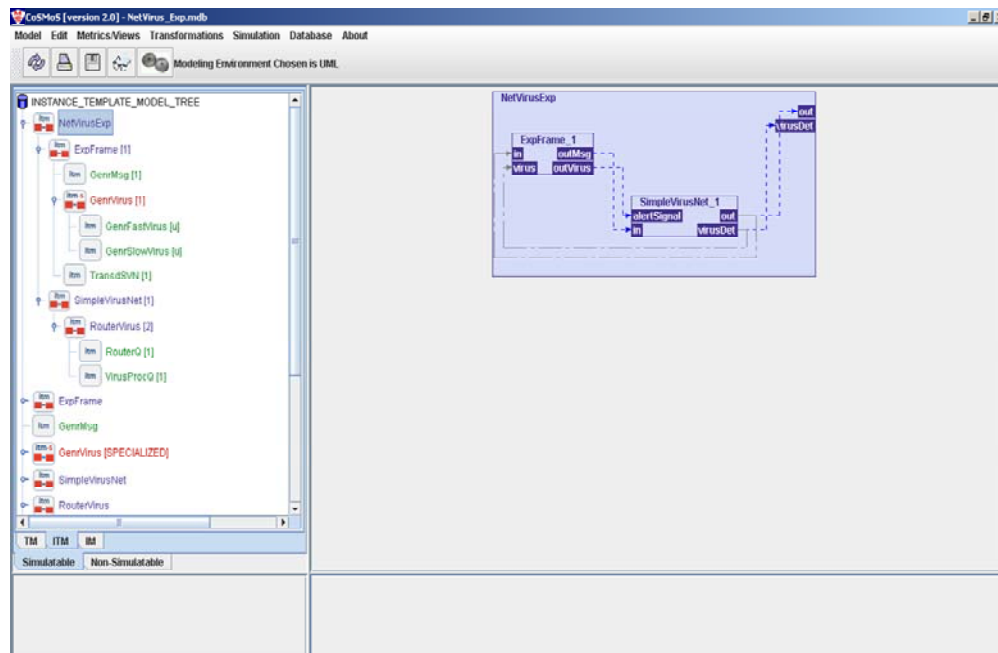


Figure 40. ITM view of the models

The Instance Template Model can be seen by selecting the *ITM* tab (Figure 40) in the *Simulatable* tab pane. The tree view shows the models with the ITM information

attached to it. The multiplicity of the components is mentioned next to the model. The graphical view of the model shows the ITM identification of the component models.

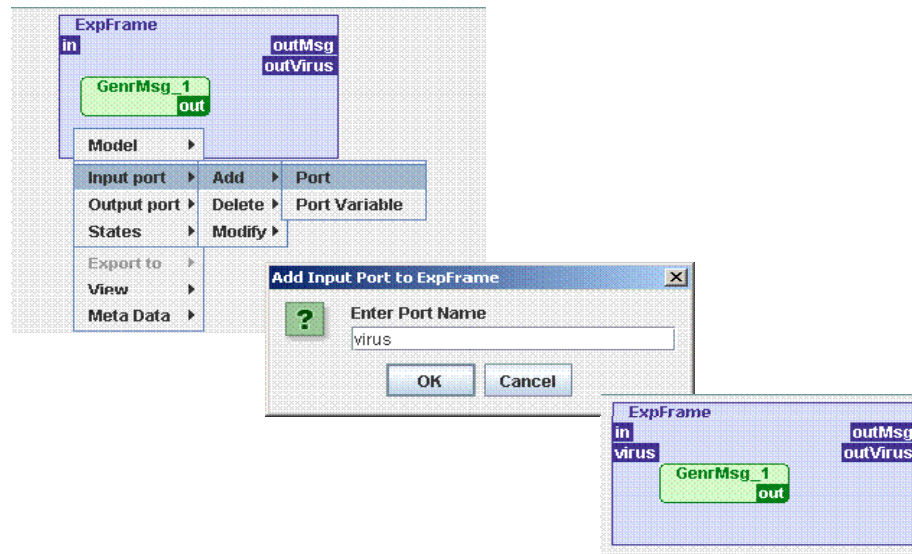


Figure 41. Adding input ports to a model

The ports have to be added to the models depending on the structure. The input/output ports for a model can be added in several different ways:

- 1) The menu for a model will pop up by right clicking on the model from the tree. The menu can also be accessed on the block model area by a right click on the model's block and following the options *Input port* → *Add* → *port*. A similar sequence of operations can be performed for the output ports. The screenshot in Figure 41 explains the process of visually adding input ports.
- 2) The option for adding the ports are also available in the menu bar, which can be invoked by ALT+F1 for input ports and ALT+F3 for output ports.

After the ports have been created, the couplings between these ports have to be built. The couplings can be established between ports by only one method. Creation of the coupling is a two-step process in which the source, i.e., *from* port, is selected and then the destination port, i.e., *to* port, is selected. This process is accomplished by right clicking within the area of the exact port and selecting the *coupling* item from the menu (Figure 42).

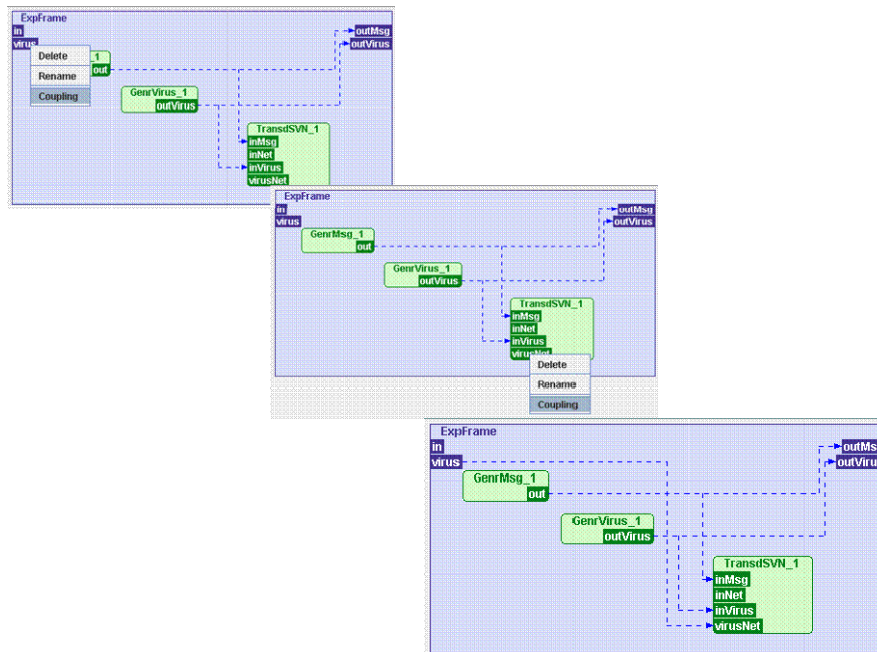


Figure 42. Adding couplings between two ports

Primitive and Composite models may also need state variables which can be added by right clicking on a model and selecting the *States* → *Add* → *State variable*. Figure 43 shows the screen shot for adding state variables to the models. The menu also includes provisions to modify or delete the state variable. The state variable accepts a name, type, and initial value as its parameters.

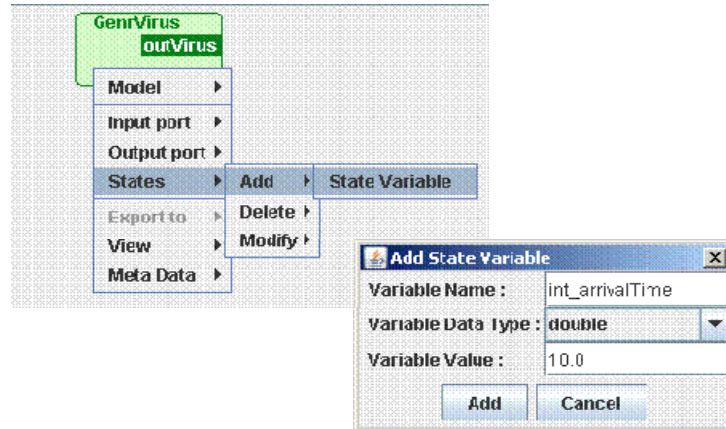


Figure 43. Adding a state variable to a model

5.1.3 Create Model Instance

After the Instance Template Models have been defined, the next step is to create the instance models, which CoSMoS allows a modeler to create from the *Model* menu. In the overall life cycle of the model, these are represented by the process of *Select instance template model* and then *create its instance models* in the CoSMo section as shown in Figure 17. The menu has an item called the *Create Model Instance*. A dialog box showing the list of all the models that can be instantiated in the current database is shown (Figure 44).

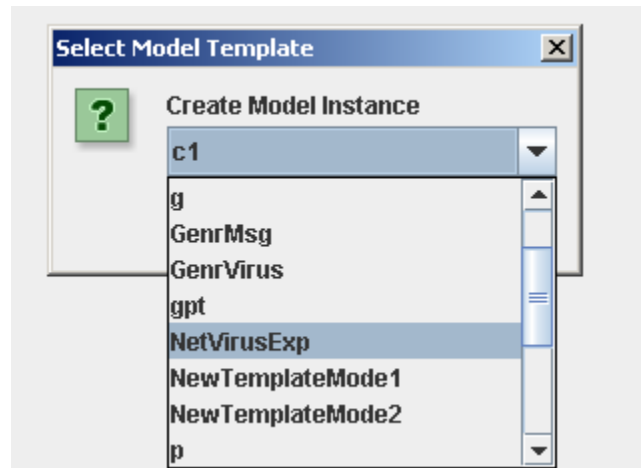


Figure 44. Creating Instance Model

The selected model forms the root of the set of the instance models to be created. The environment attaches a unique name to each of the models and its components. The environment does not allow the modeler to create a redundant instance model.

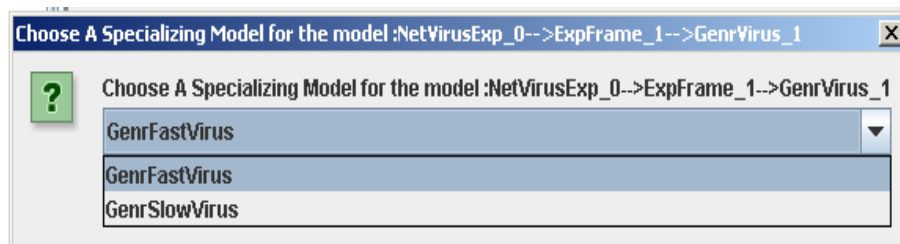


Figure 45. Choosing specialization for a specialized model

If the template model being instantiated has a model in its hierarchy that has specializations in it, a dialog box is showing the model being specialized (Figure 45).

- Name of the model being specialized.
- The model's location in the hierarchy.
- List of specializations for the model in a drop down list.

The instance models can be viewed in the *IM* tab of the *Simulatable* pane.

5.1.4 Adding Behavior

Once the instance models have been created, the models can be selected to be transformed into Java code. The transformation of these models is enabled only for the Instance Model view. The selection process is shown in the process flow in Figure 17 *Select input/output ports of models for tracking*. The model on which the transformation is initiated forms the root model. These popup menus can be made visible by either right clicking on the model in the tree or within the area enclosed by the model in the visual model layout. The model transformation is recursive as it transforms all the components in its hierarchy. The location of the generated files can be obtained by looking at the properties file accessed from the *CoSMo* menu by selecting the *View Property File*.

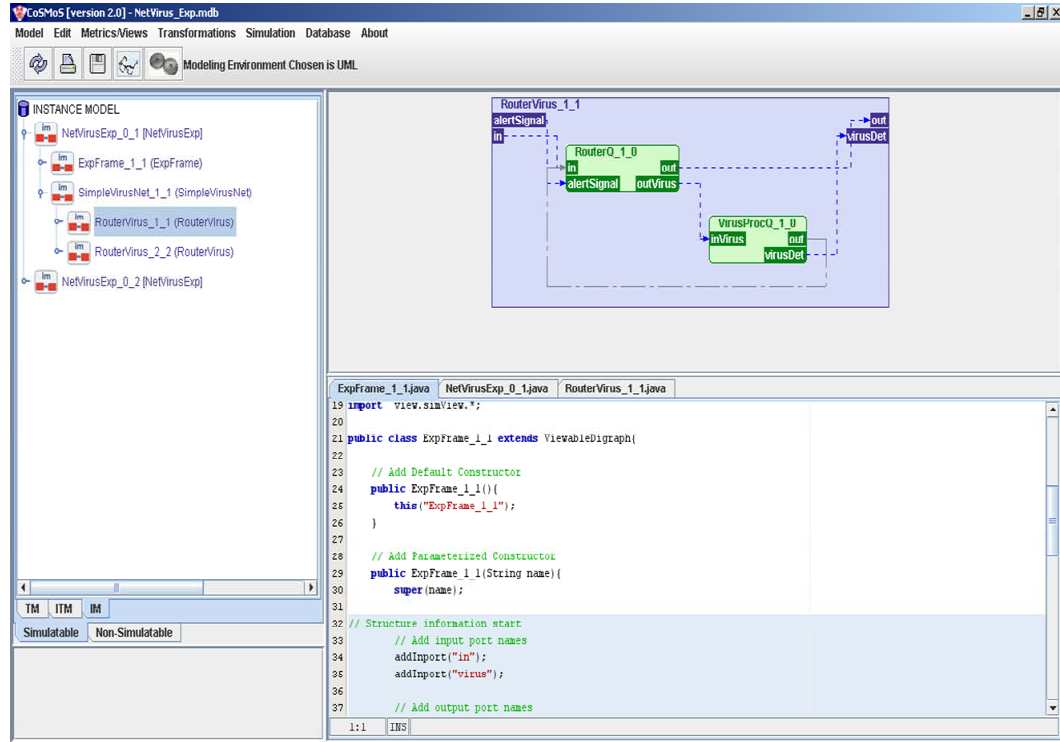


Figure 46. Adding behavior to an already exported model

The model transformation for DEVS-Suite is complete for the coupled models but not for primitive models. The models can be viewed and modified by using the CoSMoS editor. The section of the code having structural specification is locked and cannot be edited. This feature is to enforce the consistency between the model information in the database and the flat files created. Figure 46 shows the screen shot of adding behavior to the model source code using the editor built into CoSMoS.

5.1.5 Configuration

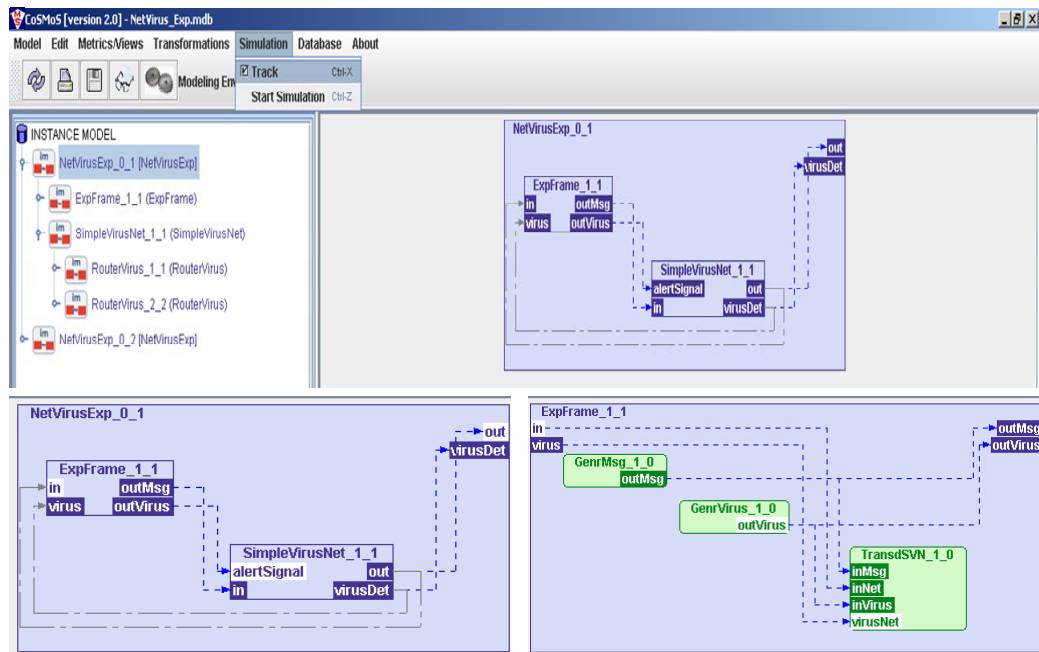


Figure 47. Selecting ports to track

Once the models are completed in terms of their behavior, they can be simulated. Now the user can determine which component is to be animated and which input/output ports will be observed. The *Track* item from the *Simulation* menu is checked to set the environment for model tracking. To set a port as tracked, the user clicks on a port and the inverted colors show that the port has been selected. The port can be unselected by clicking it again; the color of the port now returns to its original hue. The tracking process is shown in Figure 47. The environment records all the information selected by the user and uses the configuration information in the DEVS-Suite to update its tracking configuration.

5.1.6 Simulation

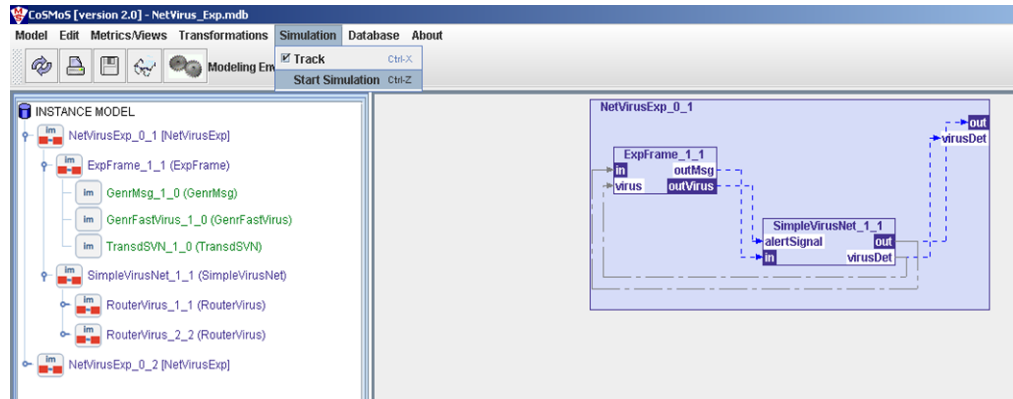


Figure 48. Loading models for simulation

After the behavior has been added into the models, they can be loaded in the simulator along with the model configuration data collected in the previous step. The Tracking mode checked during the configuration is necessary for the simulation to be enabled. Although, the model configuration can be empty, the tracking mode needs to be enabled.

The simulation process is initiated by selecting the item *Start Simulation* from the *Simulation* menu. Figure 48 shows the process of loading the models for simulation. The Java files corresponding to the models created and modified by the user are compiled against the DEVS-Suite. If the Java files are not well formed or do not pertain to the DEVS-Suite, the errors are shown in the console and must be fixed before the models can be loaded into the simulator. The code can be edited in the editor available through CoSMoS.

Once the models have been successfully loaded into the simulator, the controls are visible in the section below the tree structure of the models. Depending upon the output trajectory viewer chosen, the output of the simulation can be viewed.

5.1.7 Simulation Results

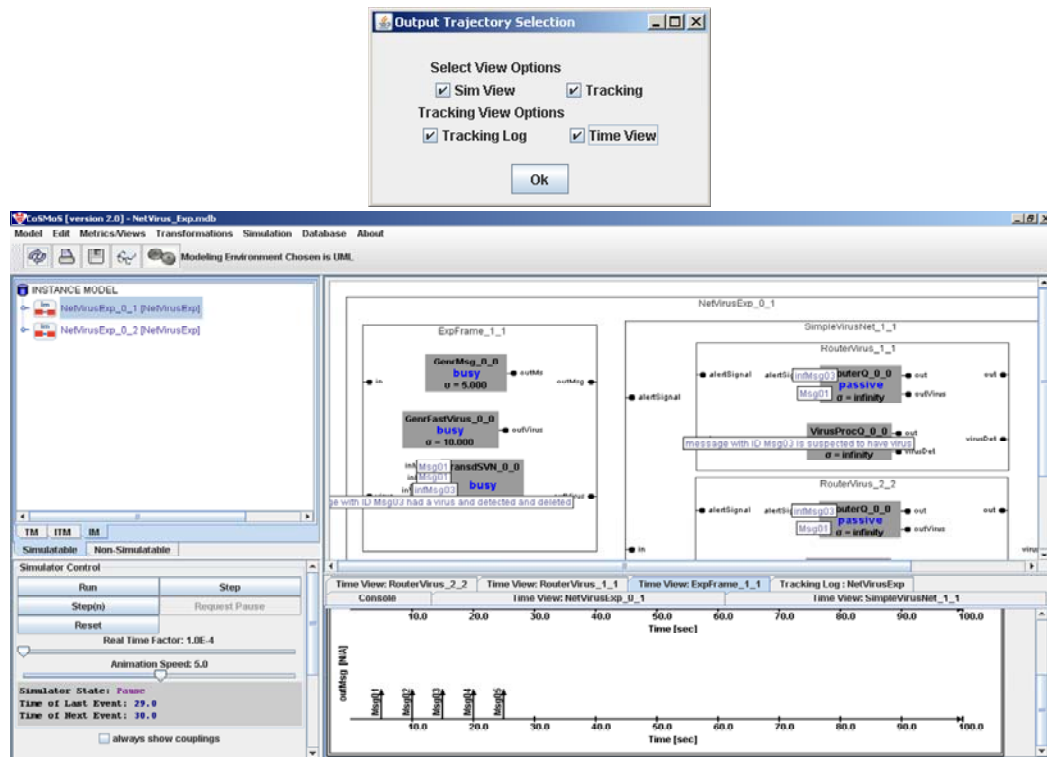


Figure 49. Options to select the output trajectory viewer

Before the models are loaded into the simulator, the user has to select the Output Trajectory viewer for the simulation data. The two major options are *SimView* and *Tracking*. The *SimView* does not use any of the tracking information that was collected as a part of the configurations process. If choosing *Tracking*, the user has to select if he or she wants the output in the *Tracking Log* or *TimeView*. Depending upon the viewer chosen, the

information is displayed in different tabs. Based upon the simulation controls, the output is updated in the trajectory viewers. The controls also have the *Real Time Factor* slider that can be adjusted to control the real time take for each simulation step. If the *SimView* is selected, the graphical representation of the model is replaced with the *SimView* animation window and *Animation Speed* slider is activated to control the speed of the animation of messages flowing between the models. Figure 49 shows the screen shot of the CoSMoS with the output trajectory viewer.

6 CONCLUSION AND FUTURE WORK

6.1 *Conclusion*

In this thesis, the CoSMo and DEVS-Suite environments are integrated following software engineering principles. The prototype CoSMoS is developed. The integrated environment CoSMoS has a clear separation among:

- Visual model development (CoSMo).
- Configuring simulation experiments (CoSMo).
- Simulating logical models (DEVS-Suite).
- Displaying simulation results (DEVS-Suite).

Developments and configuring simulation experimentation offers important capabilities within the modeling and simulation lifecycle. This environment supports creating and simulating models and thus promotes forward engineering in modeling and simulation. There exists a clear mapping for translating the generic primitive and composite models to their DEVSJAVA atomic and coupled simulation models. The mapping from the primitive model to atomic model is incomplete. The behavior for each atomic models can be added to the counterpart generic primitive models using the new editor that has been implemented. A process flow has been defined for creating the models visually in CoSMo and simulating them using the DEVS-Suite.

The work presented in this thesis serves as a basis toward developing round-trip modeling and simulation activities where, for example, model correctness can be ensured between those that are stored in database and those that can be executed using DEVS-Suite.

6.2 *Future Work*

The CoSMoS approach enables the modeler to partially automate the process of converting the models to simulation code and simulate them. The design of CoSMoS can be extended in the following areas toward greater automated modeling and simulation tool,

- 1) Behavioral modeling: The ability to visually model behavior for the atomic model needs to be introduced along with the feature of automatic behavioral model to code conversion.
- 2) Simulation code to model transformation: CoSMoS currently does not support reverse engineering of generated simulation code to database model. It is very useful to automatically update the stored primitive and composite models based on the changes made to their simulation model counterparts.
- 3) User Interface Enhancements: CoSMoS can be enhanced with respect to the user interface by allowing drag and drop feature of the models and its components.

CoSMoS currently supports the simulation of models conforming Discrete Event Specification (DEVS) since the code generated is DEVJSJAVA source code. CoSMoS can be extended in terms of its support for different models in the following areas,

- 1) Discrete-Time and Continuous system: Although the models created and simulated in CoSMoS are discrete-event models, CoSMoS may be extended to support discrete-time and continuous models.
- 2) Cellular Automata (CA) modeling and simulation. CoSMoS to provide an environment for visually modeling and constructing discrete time component based pure and composable Cellular Automata models.
- 3) Domain Specific Modeling. Using CoSMoS the modeler should be able to define or reuse domain specific models and the capability to simulate them. Some examples are Service Oriented Architecture, Network Simulation and Semiconductor Manufacturing supply chain system modeling and simulation.

REFERENCES

- ACIMS (2007). Scalable Entity Structure Modeler (SESM) (Version 1.3.0). Department of Computer Science and Engineering, Arizona State University, Tempe.
- Arizona Center for Integrative Modeling and Simulation (2007). DEVJSJAVA. <http://www.acims.arizona.edu>.
- Bendre, S. (2004). *Behavioral Model Specification Towards Simulation Validation Using Relational Databases*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Bendre, S., & Sarjoughian, H. S. (2005). *Discrete-Event Behavioral Modeling in SESM: Software Design and Implementation*, Advanced Simulation Technology Conference, San Diego, CA, USA.
- Department of EECS, U. B. (2007). Ptolemy II: <http://ptolemy.eecs.berkeley.edu/>.
- Ferayorni, A. (2008). *Domain Driven Simulation Modeling For Software Design*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Ferayorni, A., & Sarjoughian, H. S. (2007). *Domain Driven Modeling for Simulation of Software Architectures*. Summer Computer Simulation Conference, IEEE, San Diego, CA, USA.
- Fu, T. (2002). *Hierarchical Modeling of Large-Scale Systems Using Relational Databases*. Master Thesis, Department of Electrical & Computer Engineering, University of Arizona, Tucson.
- Jayadev, M. (1986). Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1), 39-65.
- Kim, S. (2008). *Simulation of service Based System: Modeling and Implementation using the DEVS-Suite*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Kim, S., H. S. Sarjoughian, R. Flasher, & V. Elamvazhuthi (in preparation). *DEVs-Suite: A Component-based Simulation Tool for Rapid Experimentation and Evaluation*. Spring Simulation Multiconference, San Diego, CA, USA.
- Lee, E. A. (2003). *Overview of the Ptolemy Project* (No. UCB/ERL M03/25), Department of Electrical and Computing Engineering, University of California, Berkeley.
- Lutz, R., Scrudder, R., & Graffagnini, J. (1998). High Level Architecture Object Model Development And Supporting Tools. *Simulation*, 71(6), 401-409.

- MathWorks (2007). How does SIMULINK perform simulations. <http://www.mathworks.com/support/solutions/data/1-15IAO.html>.
- Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., & Robbins, J. E. (2002). Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1), 2-57.
- Mohan, S. (2003). *Measuring Structural Complexities of Modular, Hierarchical Large-scale Models*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Mooney, J. (2008). *DEVS/UML - A Framework for Simulatable UML Models*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Sarjoughian, H. (2005). *A scalable component-based modeling Environment Supporting Model Validation*. 39th Interservice/Industry Training, Simulation, and Education Conference, Orlando, FL, USA.
- Sarjoughian, H. S. (2001). *An Approach for Scaleable Model Representation and Management*. Department of Computer Science and Engineering, Arizona State University, Tempe.
- Sarjoughian, H. S. (in preparation). A Unified Logical, Visual, and Persistent Component-based Modeling Framework.
- Sarjoughian, H. S., & Flasher, R. (2007). *System Modeling with Mixed Object and Data Models*. DEVS Symposium, Spring Simulation Multi-conference, Norfolk, VA, USA.
- Sarjoughian, H. S., & Singh, R. (2004). *Building Simulation Modeling Environments Using Systems Theory and Software Architecture Principles*. Advanced Simulation Technology Symposium (ASTC), Washington DC, USA.
- Singh, R., & Sarjoughian, H. S. (2003). *Software Architecture for Object-Oriented Simulation Modeling and Simulation Environments: Case Study and Approach* (No. 03-09-2003), Department of Computer Science and Engineering, Arizona State University, Tempe.
- Trygve M. H. Reenskaug. MVC XEROX PARC 1978-79, 2008, from <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>

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

APPENDIX A

NETBEANS BASED EDITOR

The NetBeans Editor API is a publicly available open source API that allows the user to use the features of the editor available in NetBeans.

It is possible to use the base class from this module as a starting point for defining your own editor kits, syntax coloring, code folding, etc. for new languages and file formats. One of the most important features of the editor being used is the *Guarded Blocks*.

Figure 50 shows the class diagram and all the components involved in the implementation of the NBEeditor document.

The class *NBEeditorLibDemoFrame* is called by CoSMo to initialize the editor and all its subsequent components.

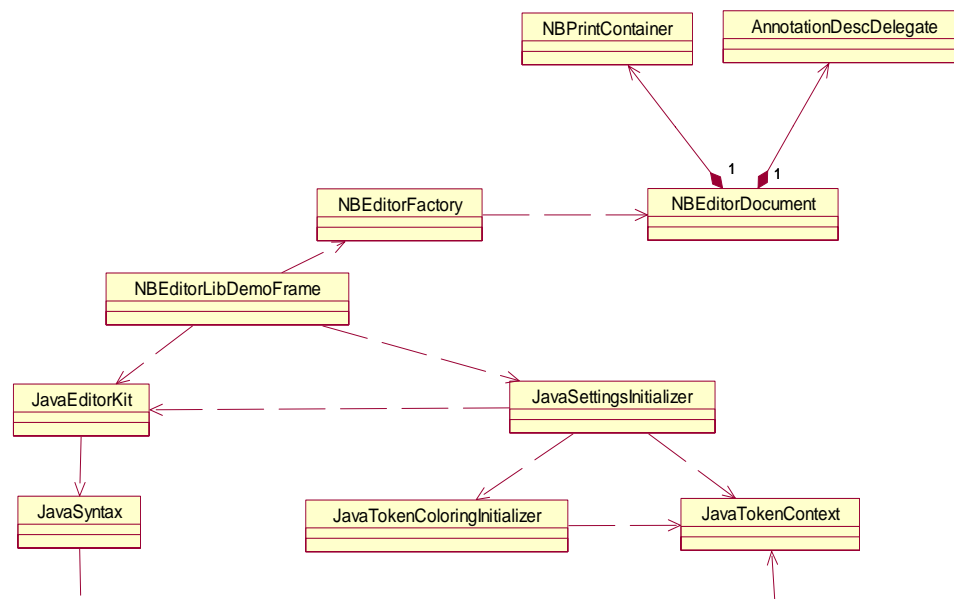


Figure 50. Class diagram of NBEeditor implementation

The *Guarded Blocks* section is an important feature of this editor with respect to CoSMoS. After the models have been completed by the user, the models should be

consistent with the database in terms of the input/output ports and couplings. Therefore the user should not be allowed to change the structural information of the individual models.

When the models are generated, they are attached with tags for the editor to identify and mark as guarded. The guarded sections are shown by the shaded region in the editor.

NetVirusExp_0_1.java	ExpFrame_1_1.java	SimpleVirusNet_1_1.java	RouterVirus_1_1.java	GenrMsg_0_0.java	GenrVirus_0_0.java
<pre> 1 /* 2 * Copyright Author 3 * (USE & RESTRICTIONS - Please read COPYRIGHT file) 4 5 * Version : XX.XX 6 * Date : 6/25/08 9:37 PM 7 */ 8 9 // Default Package 10 package MB_Models.db.JavaModels.GeneratedModelsDTE; 11 12 import java.awt.*; 13 import java.io.*; 14 import java.util.*; 15 import GenCol.*; 16 import model.modeling.*; 17 import model.simulation.*; 18 import view.modeling.*; 19 import view.simView.*; 20 21 public class gpt_0_1 extends ViewableDigraph{ 22 23 // Add Default Constructor 24 public gpt_0_1(){ 25 this("gpt_0_1"); 26 } 27 28 // Add Parameterized Constructor 29 public gpt_0_1(String name){ 30 super(name); 31 } 32 33 // Structure information start 34 // Add input port names 35 addInputPort("in"); 36 37 // Add output port names 38 addOutputPort("out"); 39 addOutputPort("result"); 40 41 // Initialize sub-components 42 ViewableAtomic g_l_0 = new g_0_0("g_l_0",10.0); 43 ViewableAtomic p_l_0 = new p_0_0("p_l_0",5.0); 44 ViewableAtomic t_l_0 = new t_0_0("t_l_0",70.0); 45 46 // Add sub-components 47 add(g_l_0); 48 add(p_l_0); 49 add(t_l_0); 50 51 // Add Couplings 52 53 // Structure information end 54 addCoupling(this,"in",g_l_0,"in"); 55 56 addCoupling(this,"start",g_l_0,"start"); 57 addCoupling(this,"stop",g_l_0,"stop"); 58 59 addCoupling(g_l_0,"out",p_l_0,"in"); 60 61 addCoupling(g_l_0,"out",t_l_0,"ariv"); 62 addCoupling(p_l_0,"out",t_l_0,"solved"); 63 addCoupling(t_l_0,"out",g_l_0,"stop"); 64 65 66 addCoupling(p_l_0,"out",this,"out"); 67 addCoupling(t_l_0,"out",this,"result"); 68 69 initialize(); 70 } 71 72 /** 73 * Automatically generated by the SimView program. 74 * Do not edit this manually, as such changes will get overwritten. 75 */ 76 public void layoutForSimView() 77 { 78 preferredSize = new Dimension(484, 176); 79 ((ViewableComponent)withName("t")).setPreferredLocation(new Point(114, 108)); 80 ((ViewableComponent)withName("g")).setPreferredLocation(new Point(7, 36)); 81 ((ViewableComponent)withName("p")).setPreferredLocation(new Point(195, 18)); 82 } 83 84 } </pre>					

Figure 51. A file generated by CoSMoS as seen in the editor