*Proceedings of the 2017 Winter Simulation Conference*
*W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.*

# RESTRAINING COMPLEXITY AND SCALE TRAITS FOR COMPONENT-BASED SIMULATION MODELS

Hessam S. Sarjoughian

Arizona Center for Integrative Modeling & Simulation
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ, 85281, USA

## ABSTRACT

From understanding our distant past to building systems of future demand having useful simulations. Standing in the way, however, is the twofold challenge of complexity and scale traits inherent to modeling and simulation lifecycle. We describe these traits in view of developing discrete event models as well as conducting experiments on them. We shed light on the concepts and methods that can help tame both structural and behavioral aspects of modeling and simulation complexity and scale needs. We demonstrate a realization of managing simulation executions and experiments in the DEVS-Suite Simulator supporting parallel component-based and cellular-based agent-based models. For developing families of system-theoretic models, we demonstrate modular, component-based structural modeling as well as state-based behavioral modeling in the Component-based System Modeler and Simulator (CoSMoS). We conclude by discussing future research directions focused on complexity and scale challenges facing heterogeneous model composability for systems of systems, cyber-physical systems, and Internet-of-Things.

## 1 INTRODUCTION

It is well known that our world is continually growing in scale and complexity. Some early prime examples include embedded software systems and computer networks which led to the Internet. Other examples are manufacturing and logistics meeting the needs of societies at large. Transportation and financial systems also have kept pace with ever more demand. In recent years, there is unprecedent ways in which engineered systems are combined with physical worlds including humans (see Figure 1). It is, therefore, not unexpected for scale and complexity of systems to have many fold increases. Of course, dependency of these first-order increases results in second and higher order increases in scale and complexity. These kinds of "connected systems" have already led to smart cities demanding driverless vehicles operating in infrastructures rich with many kinds of sensors, actuators, and computational and physical systems.

Numerous challenges face state-of-the-art principles, methods, tools, and practices necessary for understanding, analyzing, designing, implementing, evaluating, and operating such Cyber-Physical Systems. Development of sophisticated individual and collective computational, physical and natural systems depends on fundamentally new ways of thinking about modeling, simulation, model checking, verification, and validation (Davis and Bigelow 1998) (Sargent 2005, Whitner and Balci 1989). Among these, modeling is a principal barrier which affects simulation, model checking, and evaluation (see Figure 1). This is because data that underlies all models have to be formulated into structures and behaviors for natural, physical, or combinations thereof. The conceptual, mathematical, and computational representation of structure and behavior vary significantly as measured in terms of their scale and complexity traits. Generally these intertwined traits are bound to different kinds of structures and behaviors. These traits can also be ascribed to and essential to the models representing existing and futuristic systems of systems which embody Internet-of-Things, Cyber-Physical Systems, and built and natural environments.

In this paper restraining complexity and scale of systems is aimed at model development lifecycle. Software complexity and scale of simulation and model checking engines are lightly covered. Similarly complexity and scale for developing the experiments (e.g., Experimental Frame (Rozenblit 1991)) required for evaluating models (i.e., verification and validation) is examined from the vantage point of formal, visual, and persistence component-based structural and behavioral modeling applied to set-theoretic DEVS simulation (Wymore 1993, Zeigler, Praehofer and Kim 2000). These are exemplified using CoSMoS and DEVS-Suite tools.

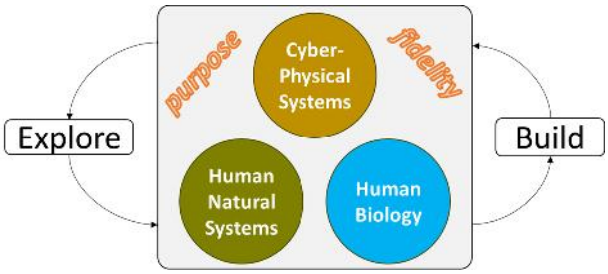



**Figure 1: Models as the core artifacts for simulation, model checking, and evaluation activities**

## 2 SCALE AND COMPLEXITY TRAITS

There exists descriptions for both scale and complexity. There are multiple meanings for scale. In the context of this paper, scale refers to sizes of some system model and any parts thereof including their relationships. As a quantitative measurement it can, for example, represent the number of parts of a model. The number of parts may also be measured as a qualitative measurement when it becomes impossible or impractical, for example, to count the number of parts or the number of ways the parts may interact with one another. Complexity broadly refers to the idea that things are difficult to understand. It is related to things being complicated and having hidden innerworkings. Computational complexity theory classifies the degree of difficulty of finding solutions to algorithms relative to their scales. More generally architectural complexity can be used to define physical, computational, and natural systems. It is useful to represent architectural complexity in terms of the following six attributes:

- Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached.
- The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.
- Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of the components – involving the internal structure of the components – from low-frequency dynamics – involving interaction among components.
- Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached.
- The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.
- Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of the components – involving the internal structure of the components – from low-frequency dynamics – involving interaction among components.

Structure and behavior of any given system are two of its fundamental characteristics. Each can be considered to have scales. Considering systems that have parts and connections, its structural scale can be easily measured. Behavior of a system may also be quantified. A system's primitive and compound operations can be considered to represent its behavioral scale. Similarly, complexity applies to both structure and behavior. Below two examples will be detailed in terms of their structures and behaviors characterized with scale and complexity traits.

As illustrated in Figure 2, structure, behavior, scale, and complexity may relate in a variety of ways to one another. Structural and behavioral scales may be quantitative or qualitative. Structural and behavioral scales are usually understood to refer to countable, natural numbers. Complexity is often viewed to be qualitative. A system can have small scale and low complexity for both its structure and behavior. Many of today's systems, however, have large scale and high complexity traits spanning both their structures and behaviors. Figure 2 shows that other possibilities exist.

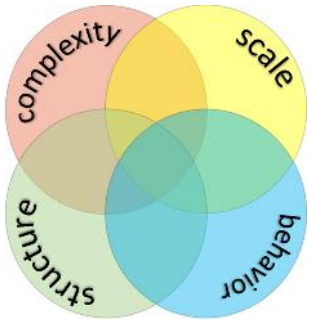



**Figure 2: System structure and behavior characterized with scale and complexity traits**

### 2.1 Structure and Behavior Scales

A sensor has a trivial structure – i.e., a glass tube filled with mercury and having a finite set of numbers printed on it. This sensor can measure temperature of some material placed in a device. It has few parts and they do not interact with another (e.g., the glass tube has no interaction with the mercury and the measurements on the tube). It has one operation. Its *structural scale* can be considered to be three. Its *behavioral scale* can be considered to be one.

A switch in a networked system has many parts (e.g., buffers and a router) with compositional (sub)structures. Each switch can have several input and output ports connected in a variety of patterns (e.g., mesh). For such systems, structural scale refers to both the number of parts and their connections. Thus the *structural scale* of the switch is high relative to that of the sensor. A switch behaves in many ways. Behavior of the switch involves those belonging to its parts. The behaviors of these parts interact with another under strict physical and time restrictions. Although parts such as buffers and router behave in limited number of ways, together they produce many kinds of behaviors. The *behavioral scale* of switch is larger than that of the sensor. The structural and behavioral scales of sensor and switch are considered to be small and large, respectively.

### 2.2 Structure and Behavior Complexities

The sensor's structure is simple. Its behavior is also simple. These observations is not surprising given the sensor's structure and behavior scales. As such sensor has trivial structural and behavioral complexity. In contrast, the switch can have a complex structure depending on its parts and their connections to one another. Complexity is low for a structure having very few kinds of parts and synthesized using basic connectors in a uniform pattern such as mesh. If on the other hand the switch has many different types of parts and they are connected in arbitrary patterns with varying kinds of connectors, complexity is high.

Behavioral complexity of the switch is complex as dispatching of packets arriving from some switches to other switches depends on conditions of the sending and receiving switches, status of the connectors between the switches. For example, if packets leave and arrive randomly and at different time instances, the some packets may have to be stored for later dispatching. As the number of switches increases, the behavioral complexity, for example, of a 10×10 network, rapidly grows. Similarly, the number and type of connectors can cause behavioral complexity to fall or rise.

The above concepts applies to systems such as coupled human-natural systems. Relatively small number of agents representing humans have complex individual and collective behaviors. Very large landscapes represented as cellular automata can have simple behaviors. In the domain of Internet-of-Things, computing platforms having tens of processors have high scale and high complexity traits. Quantifying and qualifying these traits are especially challenging to better understand, build, and operate heterogeneous systems. As such scale and complexity traits for developing heterogenous component models can be tamed using state-based and activity-based behavioral modeling methods (OMG 2004), but not in a straightforward manner using mathematical formalisms.

## 3 MACRO MODEL DEVELOPMENT AND EXECUTION LIFECYCLE

Understanding and predicting the structure and behavior of any non-trivial system of systems require having a process such as shown in Figure 3. At the core of this process is conceptualization as a collection of models (abstraction) targeted toward specific goals. Reaching each goal must eventually lead to satisfying some requirements. The models are needed for simulation and model checking. That is to say, certain models are suitable to be implemented and validated. Implementations for some other models are suitable for verification. Figure 3 shows each abstraction is necessarily limited in purpose. This can be simply understood by observing, for example any mathematical specification. Each abstraction can lend itself to one of many implementations which means the abstraction is incomplete and necessarily must be transformed to one or more other abstractions useful for implementation. Continuing with simulation and validation stages, developing other abstractions become necessary. Similarly, model checking and verification demand yet some other abstractions that can satisfy their interrelated needs. It is helpful to note modeling in Figure 3 is purely for simulation and model checking with validation and verification.

The macro level model development and execution lifecycle can be divided into two lifecycles. One has abstraction, implementation, simulation and validation. Another has abstraction, implementation, model checking and verification. For scalable, complex systems, these two processes individually and together are necessary since neither is sufficient, for exploring and building Systems of Systems.

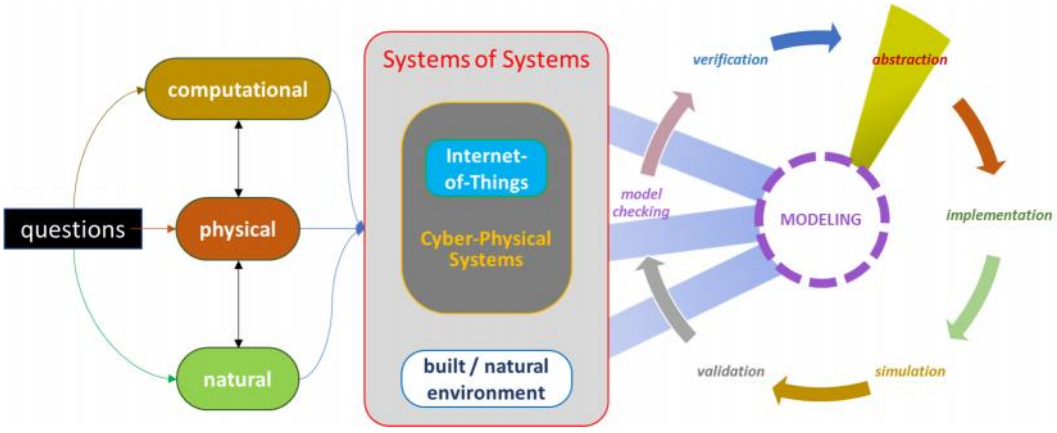



**Figure 3. Model development process cycle for simulation and model checking purposes**

- Model repositories supporting iterative, incremental structural and behavioral model specification
- Formulation of experiments and their role in simulation and model checking. Detail on model validation and verification. Brief note on accreditation of models.
- Model execution lifecycle as related to simulation and model checking algorithms. Brief note on different computing platforms.
- Scale and complexity metrics for structural and behavioral models

## 4 MICRO MODEL DEVELOPMENT LIFECYCLE

Abstractions for systems can be created using figures, formal specifications, and programming languages. These abstractions can be classified in many ways. For example, the classification shown in Figure 4 distinguishes some of the fundamental different ways of describing, specifying and implementing abstractions. It is useful to note that the kinds of methods within and across each categories may have no or some relationships to another. For example, a diagram could be a figure that has no (full- or semi-formal) syntax and semantics. On the other hand implementation of a model in a programming language can be generated from some combination of UML diagrams and DEVS formal specifications using meta-modeling Model-Driven Architecture (MDA) and the Eclipse Graphical Modeling Framework starting from conceptual to mathematical, to computational models. It is also important to note that to date there exist no complete set of modeling methods that automatically can specify arbitrary behavior of systems (e.g., CPS) with transformation from one of abstraction to another.

In effect implementations in programming languages are also models just as the conceptual and computation models are.

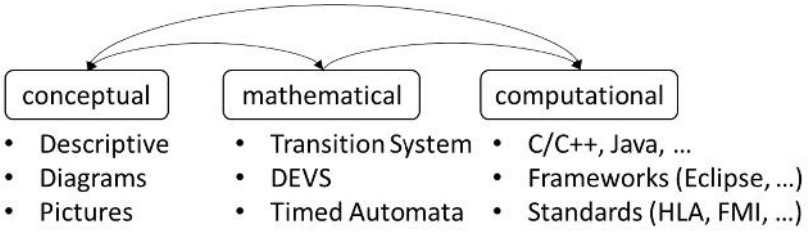



**Figure 4. A simple classification for modeling methods with frameworks**

### 4.1 Formal specifications

There exist a variety of methods for specifying behavior of dynamical systems. One early method is known as Labelled Transition System (LTS) (Keller 1976). It is specified as $\langle S, L, \rightarrow \rangle$ where $S$ is a set of states, $L$ is a set of labels, and $\rightarrow$ is a set of state transitions. The state transitions form a labeled graph consisting of $\left\{\left(p \overset{\ell}{\rightarrow} q\right)\right\}$ where $p, q \in S$ and $\ell \in L$. To account for time, Time Automaton (TA) (Alur and Dill 1994), a more expressive labeled graph, is proposed. Its specification is $\langle Q, \mathrm{K}, C, E, q_0 \rangle$ where Q is a finite set of states, K is a finite set of actions, $C$ is a finite set of clocks, $E \subseteq Q \times \mathrm{K} \times B(C) \times \wp(C) \times Q$ is a set of transitions with $B(C)$ being a set of Boolean constraints on clocks, and $q_0 \in Q$ being an initial state. Every edge $e = (q, \kappa, g, r, q') \in E$ where $\kappa \in \mathrm{K}$, $g \in B(C)$ is a guard condition, and $r$ is a clock reset. Yet another method is called Discrete Event System Specification (DEVS). This specification, unlike all other formal methods, has a concept called *elapsed time*. As a consequence, state changes are classified into the distinct external and internal transition functions. The specification $\langle X^b, S, Y^b, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle$, as in

the Labeled Transition System and Timed Automaton, allows defining state transitions. In this specification, input and output are events occurring at arbitrary monotonically time instances. There may be some bag of output events (possibly empty) $Y^b$ that can occur only after receipt of bag of input events (possibly empty) $X^b$. The external transition function ($\delta_{ext}$) is specified as $Q \times X^b \rightarrow Q$ where $Q = (s, e), s \in S, e \in ta(s)$, internal transition function ($\delta_{int}$) is specified as $Q \rightarrow Q$, the output function $\lambda$ is defined as $Q \rightarrow Y^b$, and time advance function $ta(s) \in \mathcal{R}^{+}_{0,\infty}$. A state transition can be instantaneous or take a positive finite or infinite time period. The confluent transition function $\delta_{conf}$ defines resolving concurrent external and internal transition function scheduling while guaranteeing legitimacy property. An example specification for an atomic processor model called $\text{Processor}_{\text{Queue}}$ with a FIFO queue is shown in Listing 1.

Listing1: A parallel atomic DEVS processor model with queue

$$Processor_{queue} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

where

$IPorts = \{\text{“}in\text{”}\}, where \quad X_{in} = V_X$ (an arbitrary set)

$X_M = \{(p, v) | p \in IPorts, v \in X_{in}\}$ is the set of input ports and values

$S = \{\text{“}passive\text{”}, \text{“}busy\text{”}\} \times \mathfrak{R}_0^{+\infty} \times q$

$OPorts = \{\text{“}out\text{”}\}, where \quad Y_{out} = V_Y$ (an arbitrary set)

$Y_M = \{(p, v) | p \in OPorts, v \in Y_{out}\}$ is the set of output ports and values

$\delta_{int}(phase, \sigma, q)$

$= (\text{“}passive\text{”}, \infty, q)$ if queue is empty

$= (\text{“}busy\text{”}, processingTime, q')$ otherwise remove head of the queue

$\delta_{ext}((phase, \sigma, q), e, ((\text{“}in\text{”}, x_1), (\text{“}in\text{”}, x_2), ..., (\text{“}in\text{”}, x_n))), \quad x_i \in X_{in}$

$= ((\text{“}busy\text{”}, processingTime), x_1, x_2, ..., x_n)$ if phase = “$passive$”

$= ((phase, \sigma - e), q.(x_1, x_2, ..., x_n))$ otherwise add input events to the tail of the queue

$\delta_{con}((s, ta(s)), x) = \delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(phase, \sigma, q) = (\text{“}out\text{”}, q.head), \quad q.head \in Y_{out}$, if phase = “$busy$”

$ta(phase, \sigma, q) = \sigma.$

Both Timed Automaton and DEVS are grounded in the concept of encapsulating behavior of a system within a component having strict input modularity. The concept of component in DEVS and more generally System Theory, unlike TA, also requires strict output modularity. Each of these modeling methods expressed as a *mathematical structure* has its own operational semantics which we refrain from describing here. Nevertheless, the main point to keep in mind is that such mathematical structures must be complemented with execution algorithms that can account for ordering of state transitions subject to input, state, output such that their encompassing mathematical structures are legitimate w.r.t. the cause-effect principle, concurrency, and monotonic passage of time.

A key observation to make is that encapsulation and I/O modularity are key in taming scale and complexity traits. This is possible because both the size (i.e., scale) and details (i.e., complexity) for the input, state, transition, output, and timing parts can be individually formalized. This leads to taming scale and complexity arising among these parts. In other words, the complexity of the sets, functions, and relationships that define mathematical structure can be restrained. Scale and complexity of component's structure (sets) and behavior (functions and relationships) are controlled.

The above models serve as basic parts that can be assembled to specify composite (aka as coupled and networked) models. Considering a set of LTS, they can be coupled with another as $\langle (\mathrm{A}_m)_{p \in P}, F \rangle$ where $\mathrm{A}_p$ is a finite set of LTS assigned to a set of concurrent processors $P$, and $F \subseteq \prod_{p \in P} S_p$ is a set of global final states. It is important to note that communication between LTS is abstracted to a set of channels between processors defined as $(m, n), m, n \in P$. In the DEVS formalism atomic models can be

hierarchically coupled. Such digraph models have strict tree hierarchy where every leaf node is an atomic model and all other models are coupled models. The mathematical structure for parallel coupled DEVS model $CM$ is defined as $\langle X^b, Y^b, D, \{M_{d\in D}\}, EIC, IC, EOC\rangle$ where $X^b$ and $Y^b$ are input and output event bags, $D$ is a finite set of unique names for the components contained in the $CM$, $M_{d\in D}$ the set of all unique atomic and coupled models contained in the $CM$, and $EIC$, $IC$, $EOC$ are external input coupling, internal coupling, and external output coupling, respectively. As in the parallel DEVS atomic model, behavior of coupled models is governed with its own execution algorithm in combination with that of the DEVS atomic model execution algorithm.

Returning to the scale and complexity traits, they can also be ascribed to coupled model's structure and behavior. Considering the DEVS coupled model, structure refers to its parts, inputs, and output, and hierarchical structure. Its behavior refers to its couplings as the behavior of coupled models is the result of input to input, output to input, and output to output event exchanges. The number of atomic and coupled models, levels of hierarchy, and couplings constitute structural scale and complexity. The number and frequency of event exchanges along with the content of events constitute behavioral scale and complexity.

In the following complexity and scale traits are detailed in terms of the structures and behaviors of parallel atomic and coupled DEVS models. Given the universality of DEVS relative to LTS, and DEVS, These concepts and principles apply to LTS and TA due to the relative higher expressiveness of DEVS. A system suitable for particularizing structure and behavior from scale and complexity vantage point (see Figure 2) is a hierarchical coupled model that has one processor with queue (i.e., $Processor_{queue}$ shown in Listing 1) and a flat coupled model called Experimental Frame ($EF$). This model has one Generator ($GeneratorFixed$) model and one Transducer ($Transducer$) model. The former generates tasks at fixed time intervals. The latter measures Turnaround (TA) and Throughput (TH) for the processor. The structure and behavior specified for this so-called Experimental-Frame-Processor ($EFP$) with its atomic and coupled models contain all the elements formalized for any DEVS atomic and coupled models.

### 4.2 Structural specification

The concepts of component and composition modeling are the basis for realizations of numerous frameworks and tools. These are necessary since mathematical specifications such as those in the previous section are too abstract to be automatically transformed to representations that are closer to implementation in programming languages. This can be seen by examining, for example, the DEVS formalism. Developing a simple model (e.g., EFP) quickly becomes impractical if its scale and/or complexity increases. Considering scale, it is easy to see as the number of atomic and coupled models grow, it becomes more and more difficult to know whether the model is constructed correctly. This is not unexpected since mathematical formulations do not simply lend themselves to, for example, identifying inconsistencies in a hierarchical coupled DEVS model. Furthermore, in general, it is important to develop a family of models through incremental and/or iterative steps where, for example, a model of a processor is initially developed. This model then may be specialized to two different kinds. One kind processes received tasks in a FIFO discipline while another processes the same tasks in LIFO discipline. This trivial scenario quickly overwhelms the process of model development that requires having alternative compositions for hierarchal coupled models on atomic models.

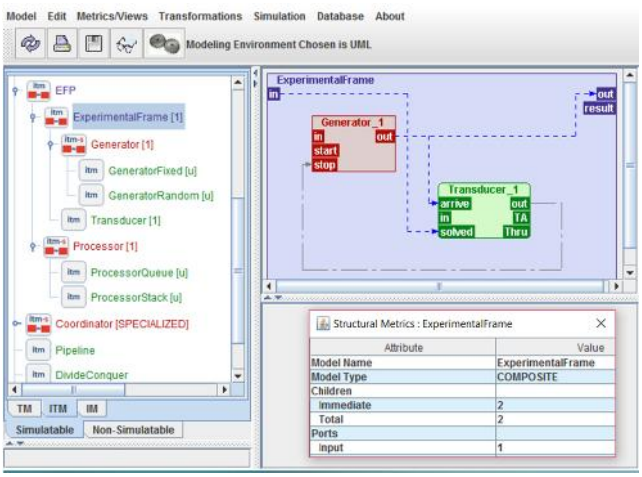
The concept of having a disciplined approach to developing families of models instead of adopting ad-hoc approaches adversely affect structural scale and complexity model development traits. It is straightforward to observe that even though modularity of atomic and coupled models components (i.e., input/output ports and couplings) is necessary, it is insufficient when specialized structures (i.e., components having alternative input/output ports and couplings and parts) is also needed. The CoSMoS unified visual, logical, and persistence modeling framework supports lessening structural model development scale and complexity traits. Both of these traits can be seen in the EFP model shown in Figure 5(a). The tree and component views together enable modelers to develop one or more kinds of EFP models. These visualized models are guaranteed to conform to the Parallel DEVS formal models. Moreover, it is

important to note that visual and persistence representations corresponding to logical representation have different structural scale and complexity characteristics. For example, as depicted in Figure 5(a), in the component view ports and couplings are straightforward to develop visually (there are no crossings of couplings, components are placed diagonally) compared with SimView in DEVS-Suite (ACIMS 2016) (see Figure 5(b)) and other similar modeling frameworks such as Anylogic (Anylogic 2017), CD++ (Wainer 2002), MS4 Me (Seo, et al. 2013), DEVS design process (Maleki, et al. 2015), and Ptolemy II (Ptolemaeus 2014). However, only one atomic model or one coupled model with its immediate parts can be seen in the component view and specializations cannot be seen, but the tree view supports viewing multiple families of models.

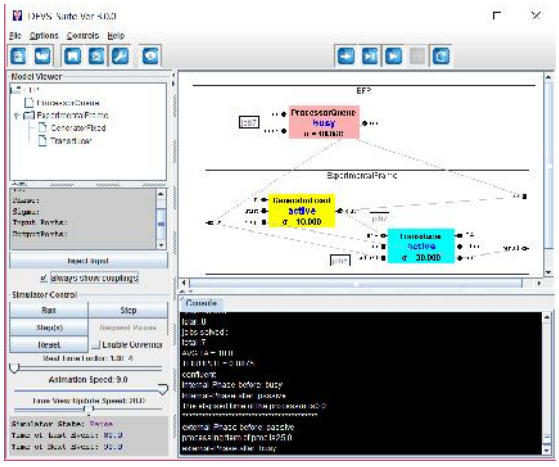
Since all models belonging to a family are individually stored in a relational database it is easy to know individual structural metrics of any component. For the ExperimentalFrame model, it has two components, three (one input and two output) ports, and four (one EIC, two IC, and one EOC) couplings. An alternative model for the $\text{Processor}_{\text{Queue}}$ can be one that can process multiple tasks simultaneously. Three basic schemes are Multiserver, Pipeline, and Divide&Conquer which are specializations of a coordinator having common input/output ports and certain functionalities such as producing outputs as shown in Figure 5(a).

Another important need for model development is to consider every concrete atomic and coupled models to have meta-models. In CoSMoS, Template Model, Instance Template Model, and Instance Model concepts are introduced. The Template Model is defined to be either primitive or composite with input/output ports. Every Composite Template Model has hierarchy of length two. The Instance Template Model allows any Composite Template Model has hierarchy of length greater than two and specifies multiplicity for Instance Models. The Instance Model defines instantiations of Instance Template Models. These models conform to system-theoretic strict modular, component-based models. For DEVS, requirements such as absence of direct-feedback in all models is checked and barred. These instance models are generated by modeler selecting specializations for both primitive and composite Instance Template Models.

The concepts underlying the CoSMoS framework are essential in developing families of Instance Models that share common Template Models as well as Instance Template Model. A close examination of the framework should reveal the role it plays in retraining scale and complexity traits far beyond developing models for target simulators and/or model-checkers. A fundamental and unique capability of the framework is that models are stored separately in database, XML, and programming code. All Instance Models are generated automatically and are guaranteed to be unique in database, XML, and implementation code. The CoSMoS tool (see Figure 5(a)) supports generating partial code for DEVS-Suite. Once the implementation code for the models are completed, they can be simulated.




(a) Component System Modeler (CoSMoS)




(b) DEVS-Suite: Simulator & Model-checker

**Figure 5. Complementary frameworks and tools for component-based modeling**

The partially generated parallel DEVS simulation models (referred to as *Simulatable* models) via the CoSMo modeling framework require having Non-Simulatable models. The essential difference between these kinds of models is that simulatable models, unlike non-simulatable, strictly conform to the DEVS formalism and execution algorithms.

The UML suite of modeling diagrams (e.g., class and component diagrams) are foundational for DEVS-Suite or other simulation engines. Example models for DEVS-Suite are bag, hashmaps, and queue. The Non-Simulatable models are integral for developing design and implementation of the elements (e.g., state set S, internal transition function $\delta_{int}$, time advance function $ta$ s and composition of hierarchical models belonging to the atomic and coupled models.

The DEVS-Suite as a simulator supports code development using IDEs such as Eclipse. The Model-Façade-View-Controller software architrave style. This simulation engine provides generic atomic and coupled models with three and component views. As in CoSMoS, it provide a kind of component view and animation. Hierarchical white/black box coupled models with atomic model components can be configured to display state information as well as animating message exchanges between any two atomic and/or coupled models. Although the simulation engine execution engine is scalable (e.g., executing many thousands of implemented models), its scale and complexity traits from the standpoint of developing families of models incrementally and iteratively (i.e., Template Model → Instance Template Model → Instance Model cycle) are weak. Although not shown in Figure 5, the simulator is equipped with TimeView supporting run-time generation of basic and superdense time trajectories through independent tracking of input and output ports for any number of atomic and coupled models (Sarjoughian and Sundaramoorthi, Superdense time trajectories for DEVS simulation models 2015). As such this simulator lends itself to retraining scale and complexity simulation model execution, experimentation and debugging.

It is also important to note that Model Driven Architecture principle lends itself quite well for further restraining scale and complexity through meta-model abstraction levels known as M3, M2, M1, and M0. A realization of the DEVS-Suite (called EMF-DEVS (Sarjoughian and Markid, EMF-DEVS modeling 2012)) for structural modeling has been proposed and developed using Eclipse different Ecore models that enforce DEVS constraints (e.g., input data type matching for the external transition function $\delta_{ext}$, detecting input/output coupling mismatches, and direct feedback). An advantage of Ecore M1 and M2 meta-models include a disciplined approach to domain-specific abstraction that are extended from their respective domain-neutral meta-models. The MDA provides a stronger basis for generating M0 models. There exist some important similarities and differences between DEVS-Suite, CoSMoS and EMF-DEVS concepts, architecture, design, implementation, and capabilities. As described above, each of these satisfies certain structural scale and complexity needs. A comprehensive discussion on the totality of these pertaining to meeting these needs, however, is beyond the scope of this paper.

### 4.3 Behavioral specification

Behaviors for dynamical systems may be defined through mathematics, programming languages, and visual notations. Each has its own benefits. Visual notation is more attractive, particularly for domain experts who may not prefer the other approaches. Coding may be preferred by those who find solving problems through trial and error. Yet other may find necessary to ground their thoughts in mathematics. Although code is the end result, as systems grow in scale and complexity, coding alone is considered insufficient since it is impractical to develop simulation models for non-trivial systems from ground up.

Considering DEVS, behavior for any atomic model is defined through $\delta_{ext}, \delta_{int}, \delta_{conf}, \lambda$ and $ta$ functions. For coupled models, behavior is defined through the $EIC$, $IC$, and $EOC$ couplings. In CoSMoS, the modelers can visually develop couplings which can be automatically translated to source code for the DEVS-Suite simulator.

The UML Statecharts and Activity diagrams are standardized visual notation for specifying a system's behavior, particularly functions such as those in atomic DEVS models. At the heart of the Statecharts is the notion of discrete states and the transitions between any two states. Parallel DEVS, unlike, UML Statecharts explicitly accounts for time. UML Statecharts, however, has a rich visual notation. Neither DEVS nor Statecharts is concerned with persistence modeling (i.e., storing and accessing models) as defined in CoSMoS. Similarly, the behavioral specifications lacking in atomic and coupled DEVS formal specifications are fulfilled using foundational UML Activity diagram provide concepts and constructs.

### 4.3.1 Statecharts specification

The CoSMoS framework is recently extended to support DEVS Statecharts (Fard and Sarjoughian 2015). It uses the Eclipse EMF, an MDA-based platform supporting independent software application development from which platform specific code can be generated. EMF distinguishes between meta-models and concrete models. A meta-model describes the structure of the concrete model. Therefore, a concrete model conforms to its meta-model. EMF framework allows creating, storing, and using the meta-model using XMI and languages including Java annotations, UML, and XML Schema. This framework is used in Graphical Modeling Framework (GMF) which provides basic entities for creating graphs consisting of nodes, links, and labels. It can be used to define special types of graphs, such as Statecharts. GMF includes wizards for generating intermediary graphical tools and mapping definitions based on an initial meta-model (EMF Ecore), as well as runtime code (R. C. Gronback 2009).

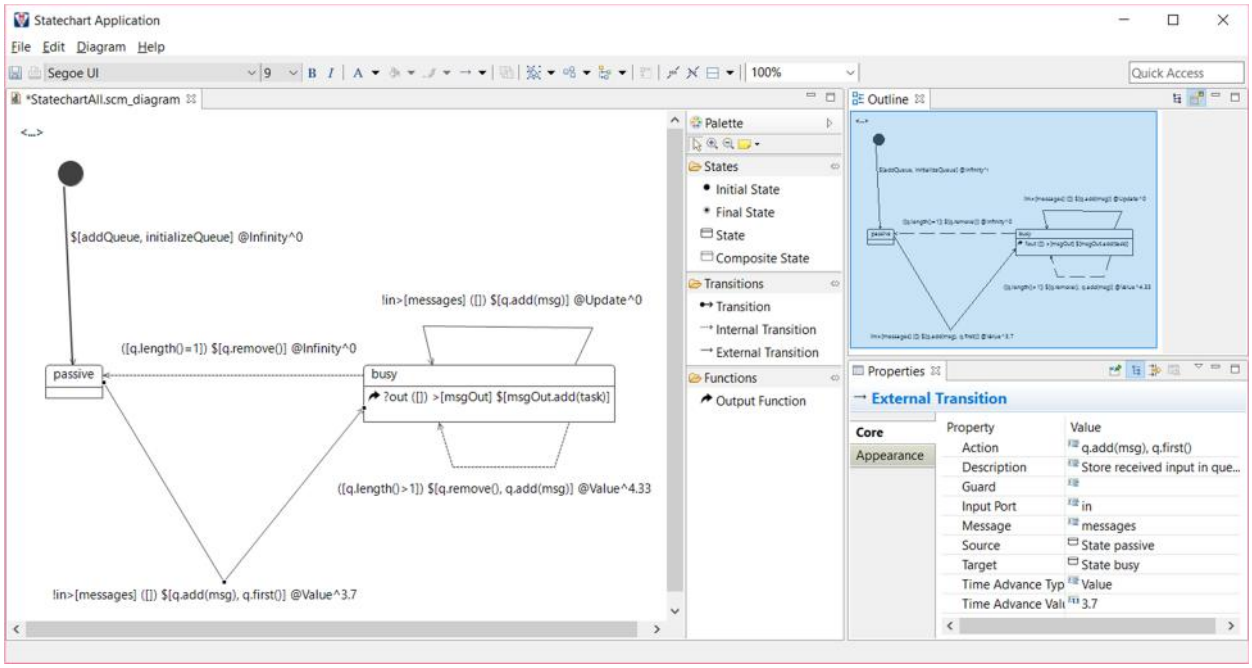



**Figure 6: DEVS Statecharts for Processor$_{Queue}$**

Time and ports are required in DEVS, but the handling of time, especially as required for simulation, has shortcomings in UML Statecharts (OMG 2004). Handling concurrent events as defined in the confluent function is not accounted for in UML. Simultaneous events are supported in UML2.0, but not in DEVS. Ports are directly handled in the UML2.0 component model. The DEVS Statecharts support modeling the functions of atomic model that can be specified in DEVS Statecharts as exemplified for the Processor$_{Queue}$. For any atomic model one or more Statecharts can be developed. For this atomic model, there are two primary state variables (aka phases). One is for distinguishing whether or not the Processor$_{Queue}$ is active

(i.e., either has a task to process) or passive (i.e., it has no task to do). When the model is in phase Passive and any number of tasks are received via input port in, they are added to queue in an arbitrary order. Then the first task is dequeued and processed and after some positive, finite time period the processes task is send out via output port out. Similarly, other self-transitions as well as transitions between phases are specified. Figure 6 shows a partial Statecharts. The semantics of the modeling elements for DEVS Statecharts satisfy those of the DEVS formalism.

The ability to develop multiple Statecharts for each atomic model can aid in restraining developing behavior from both scale and complexity aspect. This is, in part, due to being able to incrementally and iteratively specifying each of the atomic model functions. Since these Statecharts are stored in a database, scale (e.g., number of states, number of internal and external transitions) and complexity (e.g., number of actions per state transition) measurements can be determined. It is important to note that the formal specification of atomic model is too abstract for specifying details compared to the Statecharts. Thus, the level of formal specifications for atomic model functions, particularly state transitions and their relationships, are unsuitable from the standpoint of tackling scale and complexity. Since scale and complexity traits cannot be characterized, they cannot be measured and used for restraining the traits. With detailed modularized specification of the functions, behavioral scale and complexity for both atomic and coupled models can be better tamed.

### 4.3.2 Activity specification

Although Statecharts is fundamental to developing behavioral models, it is insufficient (Alshareef, Sarjoughian and Zarrin 2016). This is, in part, due to limitations in specifying ordering among actions within $\delta_{ext}$ and between $\delta_{int}$ functions as well as $\lambda$ function. Similarly, the Statecharts language is limited in terms of specifying structures such as for-loop and fork/join controls and conditional statements.

UML Activity diagram can be used to specify details of the atomic DEVS model functions and their relationships. The DEVS Activity modeling method has been developed and implemented in a similar fashion as DEVS Statecharts. The DEVS Activity modeling elements are defined based on those defined for the UML Activity diagram. Coupled model is defined using activity, expansion region, and activity edge elements among others. Atomic model is defined using activity node, Input and value pin, output pin. As in DEVS Statecharts, the DEVS Activity models have semantics that conform to those of the DEVS formalism and those that are defined for UML Activity models. In other words, UML Activity modeling is adapted and as needed extends the abstract behavior specification for the DEVS formalism.

In Figure 7, partial DEVS Activity models for the $\text{Processor}_{\text{Queue}}$ are shown. These external, internal, and output functions illustrate defining the kinds of behavior specification lacking in DEVS formalism and DEVS Statecharts. Once more behavioral scale and complexity of models are characterized in a complementary fashion. These DEVS Activity models are suited for measuring behavioral traits of atomic and coupled models which are aid in incremental and iterative model development. The resulting models are expected to have reduced scale and less complexity, for example, by eliminating unnecessary conditional statements and dependencies among actions.

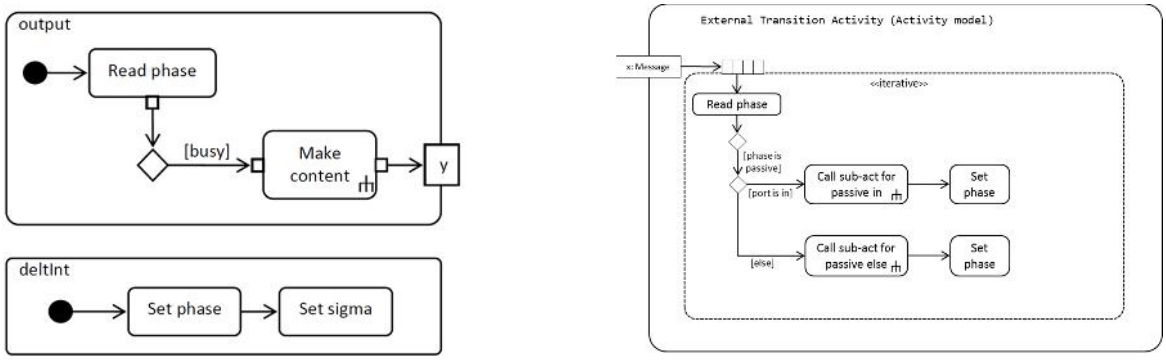



**Figure 7: DEVS Activity Modeling for state transition and output functions**

## 5 CONCLUSION

One of the fundamental challenges in model development is to strengthen linkages among conceptual, mathematical, and computational models. To address this need, this paper shows system theory, DEVS, UML, MDA, EMF, and Eclipse together are able to restrain in different, complementary ways scale and complexity inherent in developing simulation models for Systems of Systems including Cyber-Physical Systems. Restraining scale and complexity traits is achieved through partitioned structural and behavioral models. The DEVS Statecharts and Activity modeling afford details that are absent in the atomic and coupled DEVS modeling formalism. The CoSMoS and DEVS-Suite are suited for tacking scale and complexity in complementary fashion. The former supports creating families of models at component, state, and action abstraction levels while enforcing conformance to DEVS concept and formalism. Specifics belonging to functions of atomic and coupled DEVS models can be specified using DEVS Statecharts and Activity modeling. The DEVS-Suite supports executing and evaluating structural and behavioral via superdense time input, output, and state trajectories. A simple parallel DEVS coupled model is used to establish combined structural and behavioral modeling where scale and complexity traits are characterized. These DEVS concepts and principles directly lend themselves to discrete-time component modeling theory, approaches, frameworks, and tools. The foundations for scale and complexity traits inherent in structure and behavior models developed in this paper should also apply to continuous as well heterogenous models such as cellular automata, hybrid continuous-time, discrete-time, discrete-event, and other component-based models.

## ACKNOWLEDGMENTS

TBD

## REFERENCES

Alshareef, Abdurrahman, Hessam S. Sarjoughian, and Bahram Zarrin. 2016. "An approach for activity-based DEVS model specification." *Proceedings of the Symposium on Theory of Modeling & Simulation.* Society for Computer Simulation International.

Anylogic. 2017. https://www.anylogic.com/.

Davis, Paul K, and James H Bigelow. 1998. "Experiments in Multiresolution Modeling (MRM)." RAND/MR-1004-DARPA, Rand Corporation, xvii-69.

Fard, M. D., and H. S. Sarjoughian. 2015. "Visual and persistence behavior modeling for DEVS in CoSMoS." *Proceedings of the Symposium on Theory of Modeling & Simulation, SpringSim Multi-Conference.* Washington DC: Society for Computer Simulation. 227-234.

Maleki, M., R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. 2015. "Designing DEVS visual interfaces for end-user programmers." *Simulation Transactions* 91 (8): 715-734.

OMG. 2004. "Unified Modeling Language." http://www.omg.org/docs/formal/05-07-04.pdf.

Ptolemaeus, C. 2014. *System design, modeling, and simulation: using Ptolemy II.* Berkeley. http://Ptolemy.org.

R. C. Gronback. 2009. *Eclipse Modeling Project: a Domain-specific Language Toolkit.* NJ: Upper Saddle River, Addison-Wesley.

Rozenblit, Jerzy W. 1991. "Experimental frame specification methodology for hierarchical simulation modeling." *International Journal Of General System* 19 (3): 317--336.

Sargent, Robert G. 2005. "Verification and validation of simulation models." *Proceedings of the 37th Winter Simulation Conference.* 130-143.

Sarjoughian, H. S., and A. M. Markid. 2012. "EMF-DEVS modeling." *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium.* Society for Computer Simulation International.

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. "Superdense time trajectories for DEVS simulation models." *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium.* Society for Computer Simulation. 249-256.

Sarjoughian, Hessam S., Abdurrahman Alshareef, and Yonglin Lei. 2015. "Behavioral DEVS metamodeling." *Proceedings of the 2015 Winter Simulation Conference.* IEEE Press. 2788-2799.

Seo, Chungman, Bernard P. Zeigler, Robert Coop, and Doohwan Kim. 2013. "DEVS modeling and simulation methodology with MS4 Me software tool." *Theory of Modeling & Simulation-DEVS Integrative M&S Symposium.* Society for Computer Simulation International.

Wainer, Gabriel. 2002. "CD++: a toolkit to develop DEVS models." *Software: Practice and Experience* 32 (13): 1261-1306.

Whitner, Richard B., and Osman Balci. 1989. "Guidelines for selecting and using simulation model verification techniques." *Proceedings of the 21st Winter Simulation Conference.* 559-568.

Wymore, A Wayne. 1993. *Model-based Systems Engineering.* Vol. 3. Boca Raton, FL: CRC press.

Zeigler, Bernard P, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic press.

## AUTHOR BIOGRAPHIES

**HESSAM S. SARJOUGHIAN** is an Associate Professor of Computer Science and Computer Engineering in the School of Computing, Informatics, and Decision Systems Engineering (CIDSE) at Arizona State University (ASU), Tempe, AZ, and co-director of the Arizona Center for Integrative Modeling \& Simulation (ACIMS). His research interests include model theory, poly-formalism modeling, collaborative modeling, simulation for complexity science, and M\&S frameworks/tools. He is the director of the ASU Online Masters of Engineering in Modeling & Simulation program. His e-mail address is hss@gmail.com.