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
699 S. Mill Avenue
Arizona State University, Tempe, AZ, 85281, USA

ABSTRACT

From understanding our distant past to building systems of future, useful simulations demand "efficient models". Standing in the way is the twofold challenge of restraining *complexity* and *scale* of models. We describe these traits in view of component-based model development. We substantiate the roles complexity and scale play in view of modeling formalisms. We propose semi-formal modeling methods, in contrast to formal, are suitable for qualifying/quantifying model complexity and scale. For *structural abstractions*, we use class and component models. For *behavioral abstractions*, we use activity and state machines models. Furthermore, we consider these traits from the vantage point of having families of component-based models. We exemplify the concept and approach by developing families of DEVS models in the CoSMoS framework supporting DEVS-based activity and state machines models that persist in relational databases across multiple model development sessions. We conclude by discussing future research directions for real-time and heterogeneous model composability.

1 INTRODUCTION

It is well known that our world is continually growing in *scale* and *complexity*. Some early examples include embedded software systems and computer networks which led to the Internet. Other examples are manufacturing and logistics enterprises meeting the needs of societies at large. Transportation, medical, and financial systems also have kept pace with ever more demand and thus have hugely grown in scale and complexity. 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 undergone manyfold increases. Of course, dependency of these first-order increases results in second and higher order increases in system scale and complexity traits. 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 decision logics supported by computational and physical systems. Models are the essential artifacts for exploring and building such systems.

A basic goal for models of systems is that their scales are as small as possible and at the same time their complexities are minimal as much as possible. However, numerous challenges face state-of-the-art modeling principles, methods, frameworks, and practices necessary for understanding, analyzing, designing, implementing, evaluating, and operating Cyber-Physical Systems (Sztipanovits et al. 2011, Lee 2015) and more broadly, Systems of Systems (Zeigler and Sarjoughian 2012). 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. Among these, modeling is a principal barrier which affects simulation, model checking, and evaluation. Additional capabilities are needed in addition to those that have been developed over many years (e.g., (Davis and Bigelow 1998, Fujimoto 2000, Zeigler, Praehofer, and Kim 2000, Ptolemaeus 2014)). This is because concepts and information 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. Complexity and scale from the standpoint of simulation and model-checking engines are *peripherally* considered. Similarly complexity and scale for developing the experiments (e.g., Experimental Frame (Zeigler, Praehofer, and Kim 2000)) 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 DEVS-SUITE (ACIMS 2015b, Kim, Sarjoughian, and Elamvazhuthi 2009) and CoSMoS (ACIMS 2015a, Sarjoughian and Elamvazhuthi 2009) frameworks. We do not directly consider cases where the complexity and scale of simulation models undergo huge changes during execution.

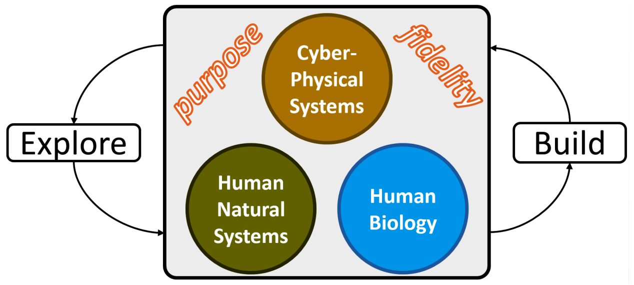


Figure 1: Models as the core artifacts for simulation, model checking, and evaluation activities.

2 Model SCALE AND COMPLEXITY TRAITS

The concepts of scale and complexity are critical in developing models for the kinds of systems shown in Figure 1. Scale and complexity are intrinsically distinct and are essential for quantifying/qualifying the power of modeling methods. As key measures of model expressiveness, every theory, methodology, and framework has scale and complexity traits. There are multiple meanings for scale. In the context of this paper, scale refers to sizes of some system model or any parts thereof including their properties, functions, and relations. 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 relate to one another. Broadly, complexity refers to the presence of hidden order in a system with some well-formed boundary. Complexity may be said to have varying degrees of complicatedness – i.e., reducing a system's complicatedness results in lessening the system's complexity (Tang and Salminen 2001). Aside from scale, increasing the kinds of parts along with the choices of their relationships directly affects a system's complexity. Computational complexity theory classifies the degree of difficulty of finding solutions to algorithms relative to their scales. Complexity is also considered for software in various ways (Denvir, Herman, and Whitty 2012). Execution time and storage required to perform computation is one kind of measurement. Another focuses on difficulty in code development, debugging, testing, and other related tasks. More generally, architectural complexity can be used to define physical, computational, and natural systems as illustrated in Figure 1. It is useful to represent architectural complexity in terms of the following six attributes (Simon 1962, Sarjoughian 2002):

- 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.
- Hierarchic systems are usually composed of only *a few different kinds of subsystems* in various combinations and arrangements.
- A complex system that works is invariably found to have evolved from *a simple system* that worked …. A complex system designed from scratch never works and cannot be "patched up" to make it work. One has to start over, beginning with a simple working system.
- *Heterogeneity* has a direct relationship with a systems level of complexity.

Structure and behavior of any given system are two of its fundamental characteristics. Both scale and complexity apply to both structure and behavior as shown in Figure 2. 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 systems primitive and compound operations can be considered to represent its behavioral scale. Below, two examples will be detailed in terms of their structures and behaviors characterized with scale & complexity traits.

As illustrated in Figure 2, *structure, behavior, scale*, and *complexity* may relate in a variety of ways to one another. A simple classification is to relate structure and behavior pairwise with scale and complexity. Structural and behavioral scale and complexity may be quantitative or qualitative measures. Considering component-based models, structural scale and complexity can be usually measured quantitatively. Behavioral scale and Complexity, however, are generally qualitative measures. A system can be small scale and have low complexity for both its structure and behavior. Many of todays systems, however, are large scale and have high complexity traits spanning both their structures and behaviors.

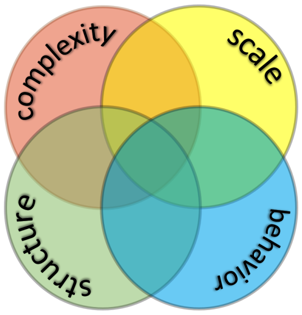


Figure 2: System structure and behavior characterized with scale and complexity traits.

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). It also has a simple behavior (i.e., measuring temperature of some material placed in a device). It has few parts and they do not interact with one 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. It is also obvious that this sensor has simple *structural complexity* as well as *behavioral complexity*.Its structure and behavior have trivial scale and complexity.

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). Each switch has many tens of parts with numerous connections. For such systems, structural scale refers to both the number of parts and their connections. Thus, the structural scale and complexity of this kind of switch is higher than that of the sensor. A switch behaves in many, often complex ways by itself and as part of a networked system. Behavioral scale 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 routers behave in a limited number of ways, together they produce many kinds of complex behaviors. Thus, structures and behaviors of switches and their composition in a network exhibit high-scale and high-complexity.

2.2 Structure and Behavior Complexities

The sensors structure is simple. Its behavior is also simple. These observations are not surprising given the sensors 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 and status of the connectors between the switches. For example, if packets leave and arrive randomly and at different time instances, then 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 heterogeneous component models can be tamed using *state-based* and *activity-based behavioral modeling* methods (OMG 2017), 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 becomes necessary. Similarly, model-checking and verification demand some other abstractions that can satisfy their interrelated needs.

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.

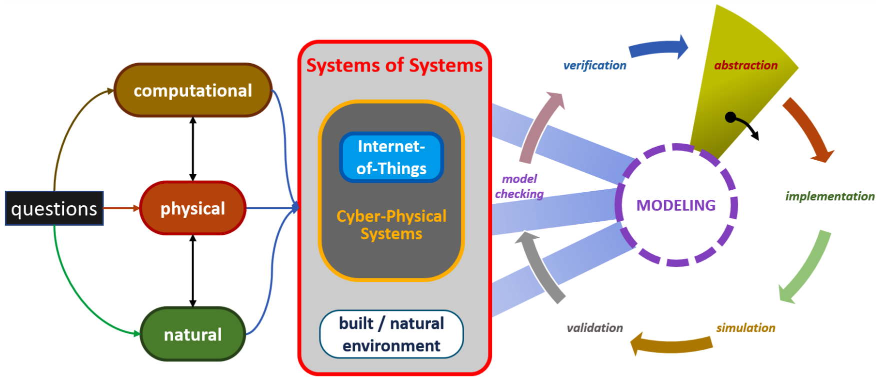


Figure 3: Model development process cycle for simulation and model checking purposes.

4 MICRO MODEL DEVELOPMENT LIFECYCLE

Abstractions for systems can be created using informal, semi-formal, and formal methods. For each there exist alternative abstractions. For example, the simple classification shown in Figure 4 distinguishes some of the fundamental different ways of conceptualizing and specifying abstractions. It is useful to note that the kinds of methods within and across each categories may have no or some direct relationships to another. For example, a diagram could be a figure that has no (semi-formal or formal) syntax and semantics. Considering formal methods, some continuous and discrete time models are composed to create hybrid mathematical abstractions. Similarly, collection of modeling methods such as UML Activity, Class, Component, and Statecharts are developed to create powerful computational abstractions. (In this paper, when the scope of discussion is more general, we use State Machines instead of Statecharts.) Furthermore, semi-formal and formal modeling methods (e.g., UML diagrams and $DEVS$) are proposed to be used using meta-modeling framework (e.g., *Model-Driven Architecture (MDA)* and the Eclipse Modeling Framework). 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., Cyber-Physical Systems) with transformation from one of abstraction to another.

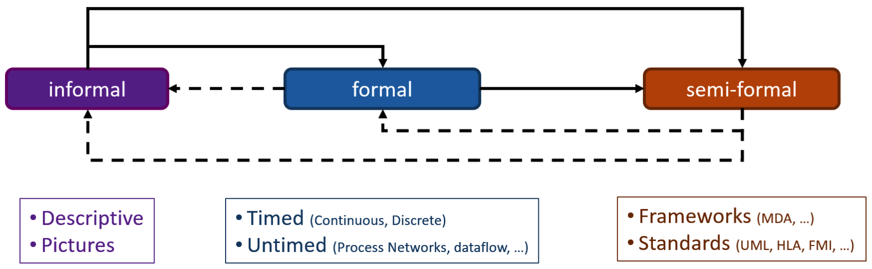

informal
formal
semi-formal
• Descriptive
• Pictures
• Timed (Continuous, Discrete)
• Untimed (Process Networks, dataflow, …)
• Frameworks (MDA, …)
• Standards (UML, HLA, FMI, …)

Figure 4: A simple classification for component-based modeling approaches.

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 $\{p \xrightarrow{\ell} q\}$ 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 $\langle Q, K, C, E, q_0 \rangle$ where Q is a finite set of states, K is a finite set of actions, C is a finite set of clocks, $E \subseteq Q \times K \times B(C) \times \wp(C) \times Q$ is a set of transitions with $B(C)$ being a set of Boolean constraints on clocks, and $q_o \in Q$ being an initial state. Every edge $e = (q, k, g, r, q') \in E$ where $k \in , g \in B(C)$ is a guard condition, and r is a clock reset.

Yet another modeling method is called Parallel Discrete Event System Specification (DEVS) (Zeigler, Praehofer, and Kim 2000). This specification and its classic predecessor, 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 monotonic 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 δ_{ext} is specified as $Q \times X^b \to Q$ where $Q = (s, e), s \in S, e \in ta(s)$, internal transition function δ_{int}) is specified as $Q \to Q$, the output function λ is defined as $Q \to Y^b$, and time advance function $ta(s) \in \Re^{+}_{[0,\infty]}$. A state transition can be instantaneous or take a positive finite or infinite time period. The confluent transition function δ_{conf} defines resolving concurrent external and internal transition function scheduling while guaranteeing the legitimacy property.

The DEVS modeling approach is grounded in the concept of encapsulating behavior of a system through modular input and output ports. That is, changes to the state of a model is either internal or only through events received through input ports. Similarly, state of a model is only accessible as events made available through output ports. Thus, in DEVS and more generally System Theory, unlike LTS and TA, an atomic model is strictly modular. Each of the LTS, TA, and DEVS modeling methods expressed as a *mathematical structure* has its own (operational) semantics which we refrain from describing here (Keller 1976, Alur and Dill 1994, Zeigler, Praehofer, and Kim 2000). 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, and output such that their encompassing mathematical structures are legitimate with respect to the cause-effect principle, concurrency, and monotonic passage of time.

A key observation to make is that encapsulation and IO 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 components 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 $\langle (A_m)_{p \in P}, F \rangle$ where A_p is a finite set of LTS assigned to a set of concurrent processors P, and $F \subseteq \Pi_{p \in P} S_p$ 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 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 atomic DEVS 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 models 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.

5 Complexity and Scale Traits of Parallel DEVS Models

Given the above modeling formalisms, we will use *DEVS* to show scale and complexity traits for atomic and composite models. We use this modeling formalism with the DEVS-SUITE modeling framework to define scale and complexity measures for DEVS-based class and component structural models. To define scale and complexity measures, we use DEVS-based Statecharts and Activity behavioral models with the COSMO modeling framework. Furthermore, we show scale and complexity traits for family of models using CoSMoS. Table 1 shows these using structural and behavioral modeling frameworks for quantifying and qualifying scale and complexity of atomic and coupled Parallel DEVS specifications.

Table 1: Structural and behavioral specification methods & frameworks for DEVS mathematical models.

Type	**Structure Specifications**	**Behavior Specifications**
	Model [FRAMEWORK]	*Model* [FRAMEWORK]
Individual Model	*Class* [DEVS-SUITE]	*Statecharts, Activity* [COSMOS]
Family of Models	*Component* [COSMOS]	*Statecharts, Activity* [COSMOS]

5.1 Experimental-Frame-Processor Model

To show complexity and scale traits for component-based models, we detail structures and behaviors of atomic and coupled Parallel DEVS models shown in Figure 5. A system suitable for exemplifying structures and behaviors from scale and complexity vantage points (see Figure 2) is a hierarchical coupled model (named `EFP`) that has one atomic model and a coupled model. The atomic model is $\texttt{Processor}_{\texttt{Queue}}$ and the coupled model is (`ExperimentalFrame`). This model has one Generator ($\texttt{Generator}_{\texttt{Fixed}}$) 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 with queue. The structure and behavior specified for this *Experimental-Frame-Processor* (named `EFP`) with its atomic and coupled models contain all the elements formalized for any DEVS atomic and coupled models. A component view of this model is developed in the DEVS-Suite (see Figure 5). The `EFP` model belongs to a family of models developed with CoSMoS (see Figure 5).

5.1.1 Processor-Queue Model

It is straightforward to define coupled DEVS models given I/O modularity and strict hierarchy. However, the same cannot be said about atomic models as exemplified with an atomic processor model called $\texttt{Processor}_{\texttt{Queue}}$ having a *FIFO queue* (see Listing 1). We developed this new mathematical model to shed light on complexity and scale traits of *atomic Parallel DEVS formalism*. This processor can handle arrival and storage of multiple input events received at any given time instance. The queue is used to store jobs when the processor is busy in processing another job. We note that the queue is one of the state variables for the $\texttt{Processor}_{\texttt{Queue}}$ model. Even though the queue has both structure and behavior, from the standpoint of the mathematical specification in Listing 1, it is abstracted to a ”state variable” having $\bar{q}, q'$ and q'' values. The queue can have no input event ($\bar{q}$), one input event (q'), or a finite number of input events (q'') at any instance of time. Received input events are added to the end (tail) of either an empty or a non-empty queue. Given an empty queue, it will have at least one input after receiving any input. Any processed input is removed from the front (head) of a non-empty queue. A queue with one input (q') becomes empty ($\bar{q}$) once the event is processed in the δ_{ext} or δ_{int} transition functions.

OBSERVATION — *The structural and behavioral artifacts included or excluded in any semi-formal and formal modeling method are determining factors for restraining scale and complexity traits.* For example, considering the `Processor`$_{\texttt{Queue}}$, we can note that operations (i.e., actions add, remove, and query) on the queue are not explicitly specified. Only the state of the queue, not the queue itself, is represented. In Parallel DEVS, the *what* state changes, but not the *how to*, are specified. The order of state changes *within* and *between* δ_{ext} and δ_{int} transition functions are unspecified. Consequently, operations such as assignments and functions needed to change the state of the model in the transitions functions cannot be explicitly specified. It should also be noted that time assignment for each of the transition functions is defined holistically (i.e., an amount of time is allocated for all the operations that together define a state change).

$Processor_{queue} = (X^b, S, Y^b, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ where
$X^b = \{(p,v) | p \in IPorts, v \in X_{in}\}$
  $IPorts = \{\text{“}in\text{”}\}$ is an arbitrary input port name
  $X_{in} = V_X$ is an arbitrary set of input variable names and values

$S = \{\text{“}passive\text{”}, \text{“}busy\text{”}\} \times \Re^{+\infty}_{>0} \times \{\bar{q}, q', q''\}$

$Y^b = \{(p,v) | p \in OPorts, v \in Y_{out}\}$
  $OPorts = \{\text{“}out\text{”}\}$ is an arbitrary output port name
  $Y_{out} = V_Y$ is an arbitrary set of out variable names and values

$\delta_{ext}((phase, \sigma, \bar{q}), e, ((\text{“}in\text{”}, x_1), \ldots, (\text{“}in\text{”}, x_n))), x_i \in X_{in}$
  $= (\text{“}busy\text{”}, processingTime, q'|q'')$ if phase is “*passive*” and queue is empty
$\delta_{ext}((phase, \sigma, q'|q''), e, ((\text{“}in\text{”}, x_1), \ldots, (\text{“}in\text{”}, x_n))), x_i \in X_{in}$
  $= (\text{“}busy\text{”}, \sigma - e, q'')$ if phase is “*busy*” and queue is not empty

$\delta_{int}(phase, \sigma, q')$
  $= (\text{“}passive\text{”}, \infty, \bar{q})$ if phase is “*busy*” and queue has one input
$\delta_{int}(phase, \sigma, q'')$
  $= (\text{“}busy\text{”}, processingTime, q'|q'')$ if phase is “*busy*” and queue more than one input

$\delta_{con}((s, ta(s)), ((\text{“}in\text{”}, x_1), \ldots, (\text{“}in\text{”}, x_n))), x_i \in X_{in}) = \delta_{ext}(\delta_{int}(s), 0, ((\text{“}in\text{”}, x_1), \ldots, (\text{“}in\text{”}, x_n))))$

$\lambda(phase, \sigma, q|q') = (\text{“}out\text{”}, y_j), y_j \in Y_{out},$ if phase = “*busy*”

$ta(phase, \sigma, q') = \infty$ if queue has one input
$ta(phase, \sigma, q'') = \Re^{<\infty}_{>0}$ if queue has more than one input

Listing 1: Atomic processor-queue DEVS model.

5.2 Structural Specifications

The concept of component-based modeling is the basis for realizations of numerous frameworks and tools. As a semi-formal approach, it complements mathematical abstractions such as those in the previous section that are too abstract to lend themselves for restraining model complexity and scale traits. This can be seen by examining, for example, the DEVS, LTS, and TA formalisms. As shown above, developing a simple model (e.g., `EFP`) quickly becomes impractical if its scale and/or complexity increase. 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 abstractions do not simply lend themselves to, for example, identifying inconsistencies in a hierarchical coupled DEVS model. It is also straightforward to see as the behaviors of `Processor`$_{\texttt{queue}}$, `Generator`$_{\texttt{Fixed}}$, and `transducer` increase in kind and variation, so does the behavior of `EFP`. Furthermore, it is necessary to develop a family of models through incremental and/or iterative steps. For example, a model of a processor without queuing may be 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 basic scenario quickly overwhelms the micro model development process shown in Figure 4.

5.3 Individual Model

The complexity and scale for structures of atomic and composite models can be quantified. *Atomic model structural scale* is represented as the number of inputs (X^b), states (S), outputs (Y^b). *Atomic model structural complexity* is associated with the types of inputs (X^b), states (S), outputs (Y^b). The `Processor`$_{\texttt{Queue}}$ has two input ports, one output port, and three states. The input and output (tasks), in contrast to state, have no structural complexity. The scale and complexity characteristic for input and output also hold for composite models. In contrast, queue has simple structural complexity. *Composite model structural scale* is represented by the number components ($D, M_{d \in D}$) and couplings (IC, EIC, EOC). For the `ExperimentalFrame` model, it has two components, three (one input and two output) ports, and four (one external input, two internal, and two external output) couplings. *Composite model structural complexity* is considered to be the formation of couplings (feedback and feed-forward) across levels of model hierarchy. The coupled `ExperimentalFrame` and `EFP` models have feed-forward and feedback couplings.

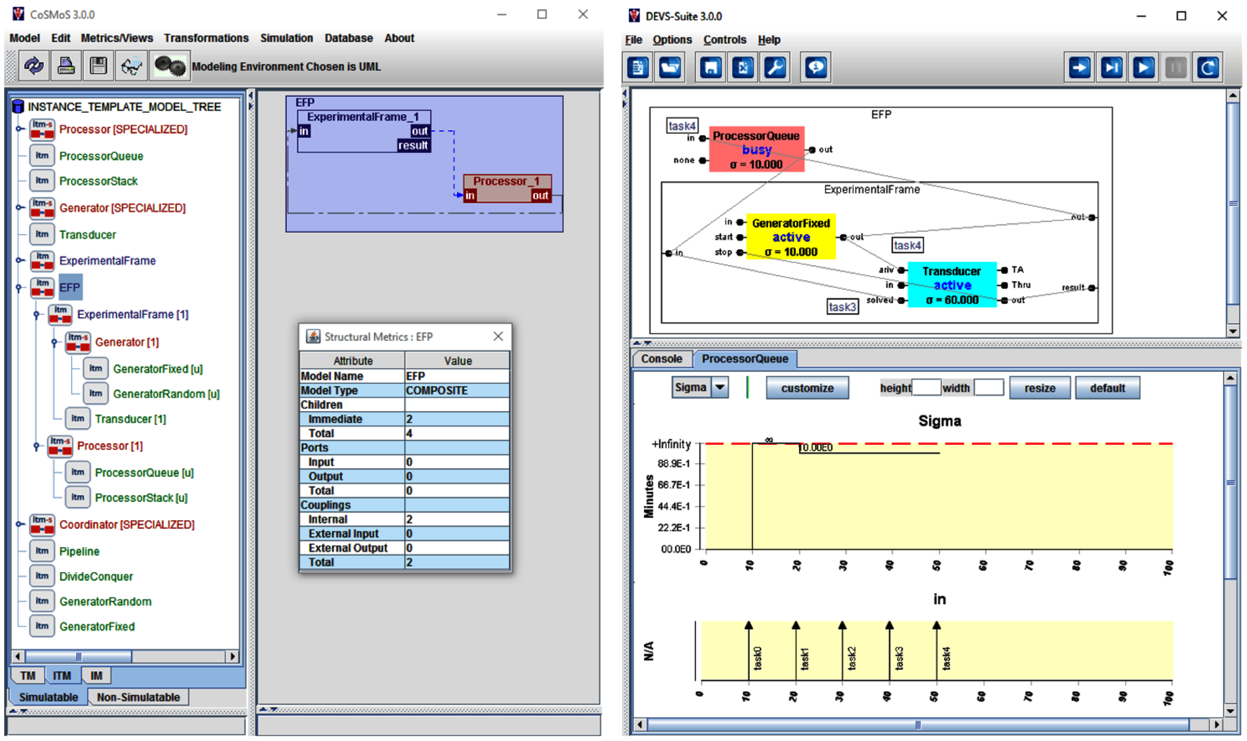
It is also useful to note that Model Driven Architecture lends itself for restraining model complexity and scale through meta-model abstraction levels known as $M3$, $M2$, $M1$, and $M0$. A realization of the DEVS-SUITE (called EMF-DEVS (Sarjoughian and Markid 2012)) for structural modeling has been proposed and developed using Eclipse different Ecore models that enforce DEVS constraints (e.g., input data type matching for the external transition function δ_{ext}, detecting input/output coupling mismatches, and direct feedback). An advantage of Ecore $M1$ and $M2$ meta-models extended from $M3$ include a disciplined approach to domain-specific abstraction extended from domain-neutral meta-models. The MDA also provides a strong basis for generating M0 concrete models. There exist some important similarities and differences in the kind of structural complexity and scale supported in DEVS-SUITE, COSMOS, and EMF-DEVS. A discussion on these and their totality for taming model complexity and scale is beyond the scope of this paper.

5.3.1 Family of Models

The concept of having a disciplined approach to developing families of models instead of adopting ad-hoc approaches is important for restraining structural scale and complexity traits of models. It is straightforward to observe that even though modularity of atomic and coupled model 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.

Toward this goal, the *Template Model*, *Instance Template Model*, and *Instance Model* concepts are introduced in the COSMOS framework. The Template Model is defined to be either primitive or composite with input/output ports. Every Composite Template Model has a hierarchy of length two. The Instance Template Model allows any Composite Template Model to have a 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 the system-theoretic modularity principle. For DEVS, requirements such as absence of direct-feedback in all models is checked and disallowed. Instance models are generated by the modeler selecting specializations for primitive and composite Instance Template Models.

The COSMOS unified *visual*, *logical*, and *persistence* modeling framework supports taming structural model development scale and complexity traits (Sarjoughian and Elamvazhuthi 2009). Both of these traits can be seen in the `EFP` model. 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



Structural Metrics : EFP

Attribute	Value
Model Name	EFP
Model Type	COMPOSITE
Children	
Immediate	2
Total	4
Ports	
Input	0
Output	0
Total	0
Couplings	
Internal	2
External Input	0
External Output	0
Total	2

Figure 5: Complementary CoSMoS and DEVS-Suite frameworks for component-based modeling.

models. Moreover, it is important to note that visual and persistence model representations corresponding to logical model representation have different structural scale and complexity characteristics. For example, as depicted in Figure 5, 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 2015b) and other modeling frameworks such as AnyLogic (Anylogic 2017), DesignDEVS (Maleki, Woodbury, Goldstein, Breslav, and Khan 2015), and Ptolemy II (Ptolemaeus 2014). However, in CoSMoS only one atomic model or one coupled model with its immediate parts can be manipulated (e.g., adding DEVS Statecharts models for any atomic model) in the component view. Specializations of atomic/coupled models as well as the family of these models is supported in the tree view.

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 relative to their XML partially generated simulation code. 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. This separation further helps with taming model complexity and scale traits. The UML modeling methods such as class and component models are foundational for the DEVS-Suite. Example, non-simulatable models are bag, hashmaps, and queue. These models are integral for design and implementation of the atomic model elements (e.g., state set S, internal transition function δ_{int}, time advance function $ta(s)$) and composition of hierarchical models.

Given the relational database for a family of models, its structural scale and complexity for atomic or composite Template Model, Instance Template Model, and Instance Model categories and all their parts can be determined. The classification of Template Model, Instance Template Model, and Instance Model with support for visual modeling and persistence is important for restraining scale and complexity of model development. *This extends taming complexity and scale for individual models to families of models.* For example, variants of the *Experimental-Frame-Processor* model known as multi-processing systems are can be developed (Zeigler, Praehofer, and Kim 2000). In particular, `Divide&Conquer`, `Pipeline`, and `Multiserver` models can be developed through specializing a `coordinator` atomic model having common input/output ports and certain functionalities such as producing outputs. As shown in Figure 5, three coordinators corresponding to the variants of the *Experimental-Frame-Processor* can be specified. The structural complexity and scale of these models are straightforward to find from the model database repository.

The DEVS-SUITE as a simulator needs an IDE such as Eclipse. Compared with COSMOS, it provides animation, but allows a restricted tree component view. 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 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 are as limited as other frameworks. A variety of factors and, in particular, choice of programming languages and communication style (creating, sending/receiving, reading, and destroying compound messages as compared with using primitive data type with global read and write access) can strengthen or weaken model abstractions. Although not shown in Figure 5, the simulator is equipped with TimeView supporting run-time generation of *linear* and *superdense time* trajectories through independent tracking of input, output, and states for any number of atomic and coupled models (Sarjoughian and Sundaramoorthi 2015). As such, this simulator lends itself to restraining scale and complexity simulation model execution, experimentation, and debugging.

5.4 Behavioral Specifications

As in structural modeling, behaviors for dynamical systems may be described using informal, semi-formal, and formal models. Each of these has its own benefits. Semi-formal visual modeling languages are attractive, particularly to the domain experts who do not prefer textual description or drawings. Yet others may prefer to ground their models in mathematics. It is useful to note behaviors for structural models are coded. However, as systems grow in scale and complexity, code and even pseudo code are not suitable means for developing abstractions.

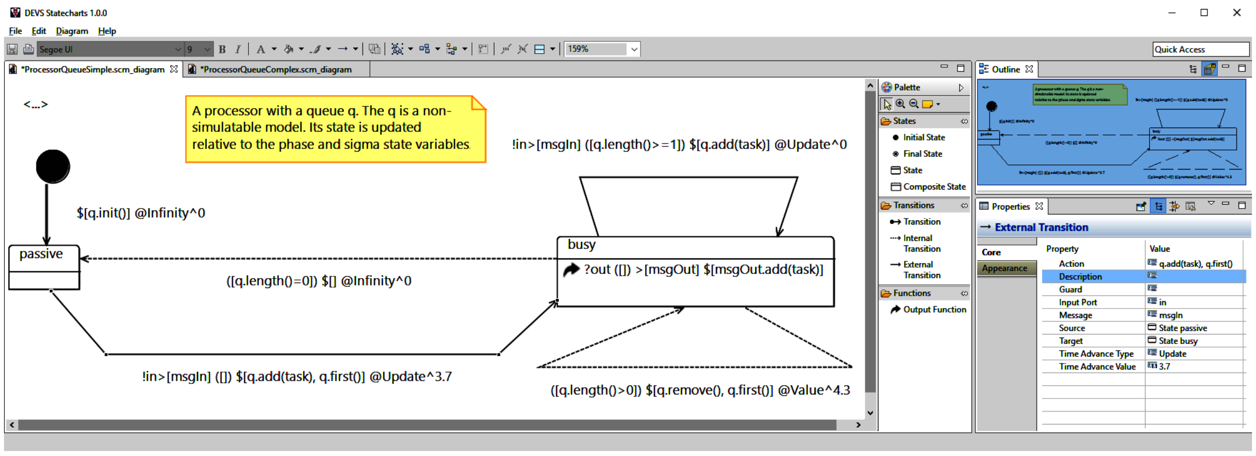
The Statecharts (a variant of the UML State Machines) and UML Activity are visual languages for modeling a systems behavior. They can be used for functions in atomic models and couplings in composite models. At the heart of the Statecharts is the notion of discrete states and the transitions between any two states. Parallel DEVS, unlike, Statecharts provides a formal basis for representing and manipulating time relative to input, output, and state transitions. Statecharts, however, provides a semi-formal visual language. Neither DEVS nor Statecharts is concerned with persistence modeling (i.e., storing and accessing models) as defined in COSMOS. The behavioral specifications lacking in atomic and coupled DEVS and Statecharts are fulfilled using Activity modeling.

5.4.1 Statecharts Specification

Considering DEVS, behavior for any atomic model is defined abstractly in terms of $\delta_{ext}, \delta_{int}, \delta_{conf}, \lambda$ and ta functions. In Statecharts, the behavior is specified in terms of specific state changes including the order in which the changes can occur. Considering δ_{ext}, any state transition for two states (S), due to arrival of input events on input ports (X^b), is specified in terms of a *guard condition*, *actions*, and *time* in Statecharts (see Figure 6). Similarly, any state transition in δ_{int} is specified in terms of a guard condition, actions, and time. Self-state transitions are allowed for both external and internal transition functions. It should be noted that formal δ_{ext} and δ_{int} do not represent actions and guard conditions. *Atomic model state-behavioral scale* is given by the number of elements of the DEVS Statecharts. *Atomic model state-behavioral complexity* is given by the number of external and internal transitions into and out of each state, the number of actions per state transition, the number of outputs allowed for each state, the number of concurrent inputs and outputs, and the number of interleaving of the internal and external state transitions.

Returning to the `Processor`$_{\texttt{Queue}}$ model, its state can be abstracted to be {"*passive*","*busy*"} instead of the 3-tuple given in Listing 1. This Statecharts is shown in Figure 6. The state of the non-simulatable queue is not represented. With this set of states, there are two external and two internal state transitions (see `ProcessorQueueSimple` Statecharts). Other behavioral artifacts such as the number of guard conditions and actions can also be known. Another abstraction for state set is {"*passiveEmptyQueue*","*busyNonEmptyQueue*"} where "*passiveEmptyQueue*" $\triangleq$ "*passive*" $\times$ "*empty*" and "*busyNotEmptyQueue*" $\triangleq$ "*busy*" $\times$ "*notEmpty*" are two composite states. For this abstraction, more state transitions are needed for each of the composite states. With this abstraction, the "*passiveEmptyQueue*" can show change from "*busy*" to "*passive*" followed by "*nonEmpty*" to "*empty*". Timing for different state-based behaviors can also be determined. Such different Statecharts reveal both behavioral complexity and scale traits of atomic models. This is, in part, due to incrementally and iteratively specifying behaviors where the `ProcessorQueueSimple` Statecharts has a smaller scale and lower complexity relative to `processorQueueComplex`. Since these Statecharts are stored in a database, it is trivial to find their scale and complexity traits.

For coupled models, behavior is defined in terms of the behaviors for all its atomic models as well as the *EIC*, *IC*, and *EOC* couplings. It should be noted that hierarchical decomposition cannot cause any changes in behavior of coupled models, although increasing model hierarchy levels adversely affects the time it takes to simulate it. *Coupled model state-behavioral scale* is directly related to those of atomic models. This is due to strict input/output modularity in DEVS and closure under coupling. This means Statecharts for atomic models that are coupled together are orthogonal to each other. *Coupled model state-behavioral complexity* increases exponentially relative to the frequency of input/input, output/input, and output/output interactions subject to the structural complexity and scale of atomic and coupled models.


DEVS Statecharts 1.0.0
File Edit Diagram Help
Segoe UI 9
159%
*ProcessorQueueSimple.scm_diagram
*ProcessorQueueComplex.scm_diagram
<...>
A processor with a queue q. The q is a non-simulatable model. Its state is updated relative to the phase and sigma state variables
!in>[msgIn] ([q.length()>=1]) $[q.add(task)] @Update^0
$[q.init()] @Infinity^0
passive
busy
?out ([]) >[msgOut] $[msgOut.add(task)]
([q.length()=0]) $[] @Infinity^0
!in>[msgIn] ([]) $[q.add(task), q.first()] @Update^3.7
([q.length()>0]) $[q.remove(), q.first()] @Value^4.3
Palette
States
Initial State
Final State
State
Composite State
Transitions
Transition
Internal Transition
External Transition
Functions
Output Function
Outline
Quick Access
Properties
Core
Appearance

External Transition

Property	Value
Action	q.add(task), q.first()
Description	
Guard	
Input Port	in
Message	msgIn
Source	State passive
Target	State busy
Time Advance Type	Update
Time Advance Value	3.7

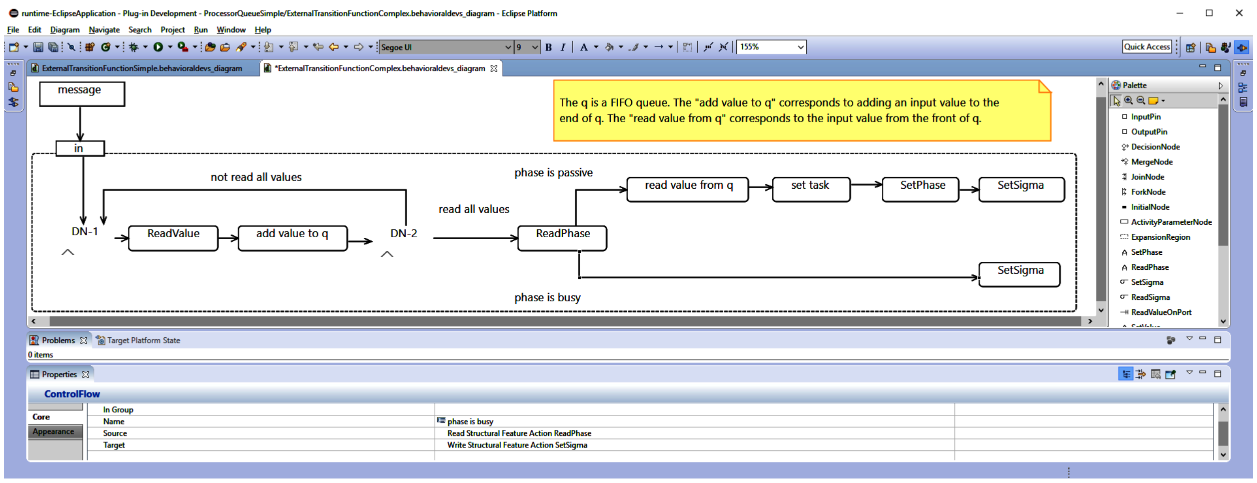
Figure 6: A DEVS Statecharts model for the DEVS atomic processor-queue model.

5.4.2 Activity Specification

Although Statecharts is fundamental for behavioral modeling, it is insufficient, in part, due to limitations in specifying ordering among actions for the δ_{ext} and δ_{int} transition functions as well as the λ and *ta* functions. Unlike Statecharts, Activity models support specifying arbitrary control structures. A DEVS-based Activity diagram can be used to specify details of the atomic Parallel DEVS model functions (Alshareef, Sarjoughian, and Zarrin 2016). Its modeling elements include action nodes, input and output pins, object and flow controls, and expansion regions. These elements are adapted from UML to conform to the semantics (e.g., receiving multiple input events at an instance of time) of Parallel DEVS formalism. Although activity modeling can be used for atomic and coupled Parallel DEVS models, it is aimed at atomic models.

Given the formalized $\texttt{Processor}_{\texttt{Queue}}$ (see Listing 1), a partial DEVS Activity model is developed for it (see Figure 7). This activity model is for the external transition function that can receive a bag of input values via the input port `in`. Using the decision node `DN-2`, all input values (`tasks`) are added to the FIFO queue `q`. The `ReadValue` and `add value to q` are action nodes. Once all the input values are stored, one of two possibilities exist. When the phase is passive, the front of the `q` is read using action node `read value from q`. Then, the `task` is processed using the action node `set task`. The changes to phase and sigma state variables are specified using the action nodes `SetPhase` and `SetSigma`. The order of these two actions are inconsequential. When the phase is busy, sigma is updated using the action node `set sigma`. The implementation of this external transition function Activity model requires fewer lines of code compared to the one shown in Listing 2. More elements, including action and decision nodes, are needed to develop the Activity model for the DEVS-SUITE code shown in Listing 2.

As in the DEVS Statecharts, DEVS activity models are useful for restraining complexity and scale traits of atomic and composite models. *Atomic model activity-behavioral scale* is given by the number of elements of DEVS Activity models. *Atomic model activity-behavioral complexity* can be given using the number of decision-making nodes for actions included in feed-forward and feedback sequences of action and other kinds of nodes. Activity models for external, internal, output, and time advance functions can have smaller scales and reduced complexity, for example, by excluding unnecessary decision and action nodes. We note that the behavior in each of the δ_{ext}, δ_{int}, and λ functions is sequential. As discussed for taming structural complexity and scale traits, as well as Statecharts, behavior of any coupled model is derived from its parts and couplings. That is, although Activity models may be used for coupled models, they do not play a central role, as they do for atomic models, in revealing complexity and scale traits, in part, given the CoSMoS modeling concepts and framework.



The q is a FIFO queue. The "add value to q" corresponds to adding an input value to the end of q. The "read value from q" corresponds to the input value from the front of q.

In Group	
Name	phase is busy
Source	Read Structural Feature Action ReadPhase
Target	Write Structural Feature Action SetSigma

Figure 7: An external transition function Activity model for the DEVS atomic processor-queue model.

```
public void deltext(double e, message x) {
/* e: elapsed time; x: a bag of input messages */
    if (phaseIs("passive")) {
      for (int i = 0; i < x.size(); i++)
        if (messageOnPort(x, "in", i)) {
          task = x.getValOnPort("in", i);
          q.add(task);
          holdIn("busy", processingTime);
        }
      task = (entity) q.first();
    }
    else if (phaseIs("busy")) {
      for (int i = 0; i < x.size(); i++)
        if (messageOnPort(x, "in", i)) {
          task = x.getValOnPort("in", i);
          q.add(task);
          sigma = sigma - e;
}   }   }
```

Listing 2: An external transition function implementation for the DEVS atomic processor-queue model.

6 CONCLUSION

Formal modeling methods are aimed at abstract representations of systems. However, the levels of abstraction in formal models, in general, do not lend themselves for knowing their complexity and scale traits. Toward overcoming this limitation, we propose complementing them with semi-formal modeling methods. Complexity and scale characteristics can be defined for semi-formal structural and behavioral models. This paper shows system-theoretic modeling, supported by MDA design approach and UML models, can afford developing component-based models that have reduced complexity and smaller scales. Such models are important for developing simulation for numerous types of systems, and more broadly Systems of Systems. Considering the DEVS modeling formalisms, models specified using it can be transformed to structural and behavioral models. In particular, the DEVS Statecharts and Activity models can represent details that are absent in the atomic and coupled DEVS models. The CoSMoS framework is well-suited for qualifying/quantifying complexity and scale of parallel DEVS models. The former supports creating families of atomic and composite component models with rich state and action specifications while conforming to the system theory and specifically the DEVS formalism. Importantly, individual elements of activity, component, and state machines models can be stored in relational databases, thus enabling evaluation of complexity and scale traits of simulation models. The latter supports model execution, thus allowing evaluation of the run-time complexity and scale of models implemented in target simulators. A simple coupled Parallel DEVS model with its atomic models is used to characterize these traits for class and component structural models as well as activity and state machines behavioral models. The foundation developed in this paper for restraining complexity and scale should be applicable to heterogeneous and real-time component-based models.

ACKNOWLEDGMENTS

This research is partially supported with NSF grants #DEB-1313727 and #CNS-1639227.

REFERENCES

ACIMS 2015a. “CoSMoS”. Available at https://acims.asu.edu/software/cosmos/.

ACIMS 2015b. “DEVS-Suite”. Available at https://acims.asu.edu/software/devs-suite/.

Alshareef, A., H. S. Sarjoughian, and B. Zarrin. 2016. “An Approach for Activity-Based DEVS Model Specification”. In *Proceedings of the Symposium on Model-Driven Approaches for Simulation Engineering*, 1–8. SCS.

Alur, R., and D. L. Dill. 1994. “A Theory of Timed Automata”. *Theoretical Computer Science* 126 (2): 183–235.

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

Davis, P. K., and J. H. Bigelow. 1998. “Experiments in Multiresolution Modeling (MRM)”. Technical report, RAND Corp Santa Monica, CA, USA.

Denvir, T., R. Herman, and R. Whitty. 2012. *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991*. Springer Science & Business Media.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. Wiley New York.

Keller, R. M. 1976. "Formal Verification of Parallel Programs". *Comm. of the ACM* 19 (7): 371–384.

Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. 2009. "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring". In *Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S*, 29–36. SCS.

Lee, E. A. 2015. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models". *Sensors* 15 (3): 4837–4869.

Maleki, M., R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. 2015. "Designing DEVS Visual Interfaces for End-User Programmers". *Simulation: Transactions of The Society for Modeling and Simulation International* 91 (8): 715–734.

OMG 2017. "Unified Modeling Language". Available at http://www.omg.org/spec/UML/.

Ptolemaeus, C. 2014. *System Design, Modeling, and Simulation: Using Ptolemy II*. http://ptolemy.org/.

Sarjoughian, H. S. 2002. "On the Role of Quality Attributes in Specifying Software/System Architecture for Intelligent Systems". In *NIST Speical Publication*, 429–434: Nat. Inst. of Standards & Technology.

Sarjoughian, H. S., and V. Elamvazhuthi. 2009. "CoSMoS: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation". In *Proceedings of the International Conference on Simulation Tools and Techniques*, 1–9. ICST.

Sarjoughian, H. S., and A. M. Markid. 2012. "EMF-DEVS Modeling". In *Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S*, 1–9. SCS.

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. "Superdense Time Trajectories for DEVS Simulation Models". In *Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S*, 249–256. SCS.

Simon, H. A. 1962. "The Architecture of Complexity". *Proceedings of the American Philosophical Society* 106 (6): 467–482.

Sztipanovits, J., X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang. 2011. "Toward a Science of Cyber-Physical System Integration". *Proceedings of the IEEE* 100 (1): 29–44.

Tang, V., and V. Salminen. 2001. "Towards a Theory of Complicatedness: Framework for Complex Systems Analysis and Design". In *International Conference on Engineering Design*, 1–8.

Wymore, A. W. 1993. *Model-based Systems Engineering*. CRC press.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Second ed. Academic press.

Zeigler, B. P., and H. S. Sarjoughian. 2012. *Guide to Modeling and Simulation of Systems of Systems*. Springer Science.

AUTHOR BIOGRAPHY

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 in the Fulton Schools of Engineering at ASU. He can be contacted at sarjoughian@asu.edu.