Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA

Bernard P. Zeigler
Hessam S. Sarjoughian

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona,
Tucson, Arizona
{zeigler,hessam}@ece.arizona.edu

Keywords:
Components, hierarchy, modularity, discrete event, system entity structure

ABSTRACT: In this paper we focus on the DEVS formalism’s support for hierarchical modular model construction and component reuse. DEVS has well defined concept of system modularity and component coupling to form composite models. It enjoys the property of closure under coupling which justifies treating coupled models as components and enables hierarchical model composition constructs. This considerably simplifies management of component models. The underlying structure of hierarchical models is captured in the DEVS concept called the system entity structure (SES). We review this concept and discuss its application to construction of combinatorially-based families of models – as most large scale model development projects are likely to generate.

Introduction

In a separate tutorial (Creating Simulations in HLA/RTI using the DEVS Modeling Framework), we describe the DEVS (Discrete Event System Specification) methodology and DEVS/HLA. The latter is an HLA-compliant modeling and simulation environment that supports high level model building using the DEVS methodology. In this paper, we focus on DEVS’ support for hierarchical modular model construction and component reuse. There are two major classes from which all user-defined models can be developed – atomic and coupled. The atomic class realizes the basic level of the DEVS formalism, while the coupled model enables coupling of DEVS model components. We show that DEVS enjoys the property of closure under coupling which justifies and enables hierarchical model composition constructs. We then show how closure under coupling also supports an important capability, namely to “componentize” a coupled model, i.e., to encapsulate it in a form that enables it to be treated as an atomic model. This considerable simplifies management of component models.

The underlying structure of hierarchical models is captured in the DEVS concept called the system entity structure (SES). We review this concept and discuss its application to construction of combinatorially-based families of models – as most large scale model development projects are likely to generate. The basic addition to the composition relation (incarnated in coupled models) is that of specialization in which alternative variants for a component can be “plugged” into an appropriate slot for it in a coupled model. Specialization is subject to interface constraints that ensure that the replacement of an abstract component by one of its variants makes sense in the encompassing coupled model. We discuss the design of such a specialization class and its implementation in DEVS/HLA with illustrations from examples developed in the DEVS/HLA (JAVA) environment.

Mathematical Systems Theory

---

1 This work was supported by Advance Simulation Technology Thrust (ASTT) DARPA Contract N6133997K-0007
Mathematical systems theory, first developed in the nineteen sixties, provides a fundamental, rigorous mathematical formalism for representing dynamical systems. There are two main, and orthogonal, aspects to the theory:

1. **levels of system specification** – these are the levels at which we can describe how systems behave and the mechanisms that make them work the way they do.

2. **systems specification formalisms** – these are the types of modeling styles, such continuous or discrete, that modelers can use to build system models.

System theory distinguishes between system *structure* (the inner constitution of a system) and *behavior* (its outer manifestation). Viewed as a black box (Figure 1) the external behavior of a system is the relationship it imposes between its input time histories and output time histories. The system’s input/output behavior consists of the pairs of data records (input time segments paired with output time segments) gathered from a real system or model. The internal structure of a system includes its state and state transition mechanism (dictating how inputs transform current states into successor states) as well as the state-to-output mapping. Knowing the system structure allows us to deduce (analyze, simulate) its behavior. Usually, the other direction (inferring structure from behavior) is not univalent – indeed, discovering a valid representation of an observed behavior is one of the key concerns of the modeling and simulation enterprise.

![Figure 1 Basic System Concepts](image1.png)

Figure 1 Basic System Concepts

An important structure concept is that of *decomposition* namely, how a system may be broken down into component systems (Figure 2). A second concept is that of *composition*, i.e., how component systems may be coupled together to form a larger system. Systems theory is *closed under composition* in that the structure and behavior of a composition of systems can be expressed in the original system theory terms. The ability to continue to compose larger and larger systems from previously constructed components leads to hierarchical construction. Closure under composition guarantees that such a composition results in a system, called its *resultant*, with well defined structure and behavior. *Modular* systems have recognized input and output ports through which all interaction with the environment occurs. They can be coupled together by coupling output ports to input ports and can have hierarchical structure in which component systems are coupled together to form larger ones. The difference between a decomposed systems, as in Figure 1 and undecomposed systems, as in Figure 2, suggests that there are distinct *levels* of systems specification – the former are at a higher level of specification than the latter since they provide more information about the structure of the system.

![Figure 2 Hierarchical System Decomposition](image2.png)

Figure 2 Hierarchical System Decomposition

The hierarchy of system specification levels is summarized in Table 1. As the level in the table increases, we move in the direction of increasing structural specificity (i.e., from behavior to structure).

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I/O Frame</td>
<td>((T, X, Y))</td>
</tr>
<tr>
<td>1</td>
<td>I/O relation observation</td>
<td>((T, X, \Omega, Y, R))</td>
</tr>
<tr>
<td>2</td>
<td>I/O function observation</td>
<td>((T, X, \Omega, Y, F))</td>
</tr>
<tr>
<td>3</td>
<td>I/O system</td>
<td>((T, X, \Omega, Y, Q, \Delta, \Lambda))</td>
</tr>
<tr>
<td>4</td>
<td>Iterative specification</td>
<td>((T, X, \Omega, Y, Q, \delta, \lambda))</td>
</tr>
<tr>
<td>5</td>
<td>Structured system specification</td>
<td>((T, X, \Omega, Y, Q, \Delta, \Lambda)) with (X, Y, Q, \Delta, \text{and} \Lambda) are structured</td>
</tr>
<tr>
<td>6</td>
<td>Non-modular coupled multi-component system</td>
<td>((T, X, \Omega, Y, Q, \Delta, \Lambda)) with (X, Y, Q, \Delta, \text{and} \Lambda) are structured</td>
</tr>
<tr>
<td>7</td>
<td>Modular coupled network of systems</td>
<td>(N = (T, X_N, Y_N, D, {M_d : d \in D}, {I_d : d \in D \cup {N}}, {Z_d : d \in D \cup {N}}))</td>
</tr>
</tbody>
</table>

Table 1 Hierarchy of System Specifications

**Relation To Object Orientation**
Models developed in a system theory paradigm bear a resemblance to concepts of object-oriented programming. Both objects and system models share a concept of internal state. However, mathematical systems are formal structures that operate on a time base while programming objects typically do not have an associated temporal semantics. Objects in typical object-oriented paradigms are not hierarchical or modular in the sense just described. The coupling concept in modular systems provides a level of delayed binding – a system model can place a value on one of its ports but the actual destination of this output is not determined until the model becomes a component in a larger system and a coupling scheme is specified. It can therefore: a) be developed and tested as a stand alone unit, b) be placed in a model repository and reactivated at will and c) reused in any applications context in which its behavior is appropriate and coupling to other components makes sense.

While coupling establishes output-to-input pathways, the systems modeler is completely free to specify how data flows along such channels. Information flow is one of many interactions that may be represented. Other interactions include physical forces and fields, material flows, monetary flows, and social transactions. The systems concept is broad enough to include the representation of any of these and supports the development of M&S environments that can make including many within the same large-scale model.

Although systems models have formal temporal and coupling features not shared by conventional objects, object-orientation does provide a supporting computational mechanism for system modeling. Indeed, there have been many object-oriented implementations of hierarchical, modular modeling systems. These demonstrate that object-oriented paradigms, particularly for distributed computing, can serve as a strong foundation to implement the modular systems paradigm.

**Systems Formalisms: Universality of DEVS**

Figure 3 depicts the basic systems modeling formalisms considered as shorthand means of delineating a particular system within a subclass of all systems. For the traditional formalisms, mathematical representation had proceeded their computerized incarnations. However, the reverse was true for the third class, the Discrete Event System Specifications (DEVS). Discrete event models were largely prisoners of their simulation language implementations or algorithmic code expressions. Indeed, there was a prevalent belief that discrete event “world views” constituted new mutant forms of simulation, unrelated to the traditional mainstream paradigms. Fortunately, that situation has changed as the benefits of discrete event abstractions in control and design became clear [1] and discrete event distributed simulation under HLA takes hold. With respect to such alternative discrete event formalisms, DEVS has been shown to be universal in the sense that any discrete event behavior can be expressed with it [2].

![System Specification Formalisms](image)

**Modularity and Coupling in DEVS**

We provide an informal exposition of the DEVS formalism, focusing on the concepts of modularity and component coupling. Implicit in the presentation will also be the levels of system specification. More detail is available in many references (e.g., [2] also see URL: www-ais.ece.arizona.edu). A DEVS model is a modular system receiving inputs, changing states, and generating outputs over a time base. Input and output streams are time-indexed series of events. Figure 4 illustrates an input event segment containing inputs $x_0, x_1, x_2$ arriving at times $t_0, t_1, t_2$ respectively; and a output event segment with outputs $y_0, y_1$ departing at times $t'_0, t'_1$ respectively. There are no constraints on the spacing between events; however, only a finite number of events are allowed in a finite time interval.

![Input and Output Event Streams](image)
coupled. An atomic model directly specifies the system’s response to events on its input ports, state transitions, and generation of events on its output ports. A coupled model is a composition of DEVS models that presents the same external interfaces as do atomic models. For example, in Figure 5 CM is a coupled model with four components. A coupled model specifies three types of coupling:

- **external input** – from the input ports of the coupled model to the input ports of the components (e.g., from start of CM to start of counter)
- **internal** – from the output ports of components to input ports of other components (e.g., from explosion of bomb to strike of target), and
- **external output** – from the output ports of components to output ports of the coupled model (e.g., from damage of target to damage of CM).

Although arbitrary fan-out and fan-in of coupling is allowed, no self-loops are permitted. DEVS is closed under coupling, which means that a coupled model can itself be a component within a higher level coupled model, leading to hierarchical, modular model construction.

Although arbitrary fan-out and fan-in of coupling is allowed, no self-loops are permitted. DEVS is closed under coupling, which means that a coupled model can itself be a component within a higher level coupled model, leading to hierarchical, modular model construction.

To illustrate the DEVS modeling process, consider a scenario in which a bomb can threaten some objects in its vicinity called targets. If the bomb is left undisturbed over a long period it will lose its potency. If its fuse is ignited, it will in a very much shorter time, explode and impact the targets. However, if during this period, it is deactivated, the bomb will not go off but return to its dormant state. Many processes have similar conditionally timed behavior. For example, a grain can be dormant until picked up by a bee and transported to pollinate other flowers; but the bee may be killed before arriving at its destination thereby aborting the attempt at plant reproduction. A preemptive processor can be leave the job currently being done if interrupted by one of higher priority. A reliable network packet protocol waits for a fixed time for all packets in a message to be acknowledged by the receiver; otherwise it retransmits them (often called a timeout). The modular, state-based concept of DEVS enables it to respond such external inputs based on its current state and the time that has elapsed in that state.

![Figure 6 DEVS Atomic Model](image)

To model this type of behavior in DEVS we define a coupled model, as in Figure 5 containing components for the bomb, some targets, and a defuser capable of deactivating the bomb. Each component is specified as a modular atomic model with input and output ports. In particular, the bomb has input ports for fusing (activating) and defusing (deactivating) and an output port for the explosion event (Figure 6).

Due to its modularity, DEVS distinguishes between internal and external events. While internal events represent state changes that are self induced, external input events, on the other hand, are impressed from outside the model and are inputs to which the system must respond. For example, the arrival of a fuse event causes the phase to change from dormant to live. As part of the effect of an external input event, the timing of the next internal event may be changed. In this case, if the time advance associated with phase live is $t$, which means that in $t$ units the phase is scheduled to change to explode (whereas before the external event, the dormant-to-dead transition was scheduled at some much later time). Output external events are generated by a system at its internal events. For example, the transition from explode to dead generates the explosion output event. Note that from the bomb’s point of view, the arrival of the defuse input is unpredictable, it could arrive in time (i.e., before the time elapses in phase live) or after this time has elapsed (including never). The DEVS formalism, which modular concept of external events, requires and allows explicit specification of the dynamics of such conditionally timed events.

![Figure 5 DEVS Coupled Model](image)

Figure 7a) illustrates a phase transition diagram that portrays the dynamics of an atomic model. The two kinds of transitions are shown – external, an arrow labeled by an internal port, and internal – a dashed arrow labeled with a time advance. For example, the transition from dormant to live is externally induced by the fuse external event. The transition from live to explode is an internal transition as is the transition from explode to dead.
The External Transition Function specifies how the system responds to external events. In general, this response is a change in state conditioned on the nature of the external event (input port of arrival and parameters of the received message), the current state, and the time elapsed since the last event. For example, an arrival on port fuse changes the phase from dormant to live. But the time that the system has been dormant might degrade the power of the explosion. This might be modeled by including a state variable, potency, that would be reset by the external transition function to a decaying exponential function of its original value and the elapsed time in phase dormant.

The Internal Transition Function specifies the change in state that occurs upon an internal event. It is given as a function that maps the state before the event to the state that is to be in effect immediately after it.

The Output Function specifies the output port and value that will be generated just before an internal event. It is given as a function that maps the state before the event to the output space (set of possible outputs). Not all states need to generate outputs. For example, transitioning from live to explode generates the explosion output, while no other transitions generate output events.

One characteristic DEVS feature that Figure 7b) does not show is the time advance function. This assigns to every state the time in which the system will stay in this state before an internal event occurs. Rather than give this function explicitly, we allow a state variable, called sigma, to hold its value for the current state, and to have this value set by the external and internal transition functions. The command “hold in (live, 10)” means set the phase to live and sigma to 10. The command “passivate in (dead)” means to set phase to dead and sigma to infinity. In other words, the system is scheduled to remain in dead forever. It is also possible for a state to have a zero time advance. For example, we might change the time to go from explode to dead (the explosion time) from 1 to 0.

Closure Under Coupling

An important feature of the DEVS formalism is that, like its systems superset, it closed under coupling [Figure 8]. This means that the resultant of a coupled model consisting of DEVS components is itself expressible within the DEVS formalism. Such closure supports the ability to “componentize” a coupled model, i.e., to encapsulate it in a form that enables it to be treated as an atomic model. This closure property further guarantees that coupled models can be included as components in more inclusive coupled models thus supporting stagewise hierarchical construction. This considerable simplifies development, verification and management of component models.

System Entity Structure

Hierarchical construction is illustrated in Figure 9, where, from a traffic perspective, a city may be decomposed into various road networks that meet at junctions where traffic can move from one to the other.

---

2 unless an external event changes this), but an event capable doing this is not shown in Error! Reference source not found.
Each road network may consist of several segments, where each segment has a traffic light and is, itself, decomposed into several segments between traffic lights. With a road segment we also associate a traffic generator to provide a local source of traffic to a road segment and a scheduler to control the generator (for normal vs. rush hour conditions) and the traffic light.

The system entity structure (SES) formalism provides an operational language for specifying such hierarchical structures. An SES is a structural knowledge representation scheme that systematically organizes a family of possible structures of a system. Such a family characterizes decomposition, coupling, and taxonomic relationships among entities. An entity represents a real world object. The decomposition of an entity concerns how it may be broken down into subentities. As already mentioned, coupling specifications tell how sub-entities may be coupled together to reconstitute the entity. The taxonomic relationship concerns admissible variants of an entity.

**Figure 9 A System Entity Structure**

As shown in Figure 9 an SES is represented as a labeled tree with attached attributes that satisfies, at minimum, the following axioms:

**alternating entity/aspect or entity/specialization:** Each node has a mode that is either entity/aspect or entity/specialization such that a node and its successors are always opposite modes; the mode of the root is entity.

**uniformity:** Any two nodes with the same names have identical attached variable types and isomorphic sub-trees.

**strict hierarchy:** No label appears more than once down any path of the tree.

The system entity structure in Figure 9 illustrates the different nodes in the structure:

- **multiple entity** (three parallel lines), showing decomposition into one or more subentities (e.g., roadNet),
- **aspect** (branching node), showing decomposition into a finite number of different entities (e.g., the decomposition under roadSeg) and
- **specialization** (paired lines), showing alternative choices for the entity (e.g., we can use the time stepped or event-based models of vehicles flowing within a segment).

In DEVSJAVA, a multiple entity is most naturally implemented as a block model with its components (which can themselves be block models as we have seen) indexed by three-dimensional addresses to augment the basic port-based coupling.

**System Entity-Structure/Model-Base (SES/MB) Framework**

The SES/MB framework [3-5] is a powerful means to support the plan-generate-evaluate paradigm in systems design. Within the framework, entity structures organize models in model base. Thus, modeling activity within the framework consists of three sub-activities: specification of model composition structure, specification of model behavior, and synthesis of a simulation model.

**Figure 10** shows a modeling and simulation methodology based on the framework in the process of iterative systems design. In the figure, the generation phase consists of two sub-phases: pruning and model synthesis. The structure specification and/or the behavior specification may already exist in the entity structure base and/or model base. However, if the structure specification is not in the entity structure base, we need to specify it by building an SES which represents a family of possible model structures. Likewise, if the desired model components are not in the model base, we need to develop them and store them in the model base for later use. . In the pruning phase, we select a sub-structure by pruning the SES with respect to design objectives. A simulation model is automatically synthesized from such a pruned entity structure. Simulation experiments may require changes
of structure and/or behavior of the design model. The pruning-synthesis-evaluation process is repeated until a desired design is found. Once simulation experiments are completed, the designer can save structure and behavior specifications in the system entity structure base and the model base, respectively for later use.

![Design Methodology Using SES/MB Framework](image)

**Figure 10 Design Methodology Using SES/MB Framework**

### Automatic Pruning of an SES

Instead of having the user prune an SES for a desired model structure, we can provide an automatic means of iterating through all prunings [6]. Provided that the number of prunings is not too large, this provides an automated search capability for finding a best design. If the number of design alternatives is too large for an exhaustive search, we have to turn to more natural, and artificial, intelligence to constrain the search space. To provide such search capability we need an algorithm that, given an SES, is capable of computing its total number of prunings and iterating through them one-at-a-time, each time synthesizing the associated hierarchical simulation model from the model base and evaluating it. The design of such an algorithm is based on the fact that an SES is built recursively with alternating specializations and decompositions. Likewise, the number of alternative prunings can be enumerated and generated recursively. The number of alternatives under an aspect is the product of the alternatives under its entities. The number of alternatives under a specialization is the sum of those under its entities. This recursion stops when leaf entities are encountered. The number of alternatives represented by a leaf entity is just 1 (itself). Finally, the number of prunings of the SES is the number of alternatives under the root node.

### Implementing SES in DEVS/HLA(JAVA)

In DEVS/HLA(JAVA), the basic DEVS formalism and the coupled model formalism are encoded in respective classes. The SES capability is obtained by extending the coupled class to a class which treats members as alternatives in a specialization rather than simultaneously coexisting components as in its parent class. For example, in Figure 11 there are two specializations of a processor, one with, and the other without, a buffer to queue the arriving jobs. Class procQ (with queue) is derived from class proc and they both have the same I/O ports. Thus they are proper alternative choices in a processor specialization.

![Experimental frame coupled to processor specialization](image)

**Figure 11 Experimental frame coupled to processor specialization**

To be of use, such a specialization must be able to replace a component in another coupled model. For example, we could have it coupled to an experimental frame as in Figure 11. We call class efp *prunable* since it has within it (at least) one specialization. By selecting one of the two choices in procSpec, we get a new model that is pruned and simulatable.
The methodology supported by DEVS/HLA(JAVA) is illustrated in Figure 12. After pruning and transforming, the modeler can immediately experiment with the resulting model. Alternatively, s/he can issue a command to write the model description into an automatically created class file. This enables reuse of the pruned structure as an executable simulation model and obviates the need to prune the master structure again to get the same model. Reuse of pruned structures adds a second layer of reuse to that of initially constructed components.

Families of Models: Constraints on Coupling

Constraints on coupling assure that any source-generated value can be handled by the receiving component. In DEVS/HLA, such values are instances of classes derived from the base class. Since such objects must be manipulated by the receiver, the constraints translate into the requirement that the receiver should be able to apply any method it needs to the instances. As illustrated in Figure 13, classes may be associated with ports. Associating a class c with output port p, indicates that values in contents of messages sent on port p are instances of class c. Similarly, associating a class with an input port means that the receiver expects to treat values arriving on this port as instances of this class.

Thus, we can couple an output port port0 with class0 to an input port port1 with class1 if every method in class1 is also in class0. Typically this is guaranteed by inheritance from class1 to class0 (yes, this is the right direction!). For example in Java, this is satisfied if class0 is derived from class1 and the methods are public. As an important special case, the requirements are satisfied if class0 and class1 are actually the same class. The following table lists all admissible combinations for the inheritance hierarchy in Figure 13. For example, if the receiver expects an entity (base class instance) than any of the derived class instances can be sent to it. If class1 is document than class0 can't be entity, but can be document or its derived classes. Finally, if the receiver expects a text document then only a text instance can be sent to it.

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>some applicable</th>
<th>some methods not</th>
</tr>
</thead>
<tbody>
<tr>
<td>class0</td>
<td>class1</td>
<td>methods at receiver</td>
<td>applicable at receiver</td>
</tr>
<tr>
<td>entity</td>
<td>entity</td>
<td>get_name()</td>
<td></td>
</tr>
<tr>
<td>document</td>
<td>entity</td>
<td>get_name()</td>
<td>get_date()</td>
</tr>
<tr>
<td>image</td>
<td>entity</td>
<td>get_name()</td>
<td>decode()</td>
</tr>
<tr>
<td>text</td>
<td>entity</td>
<td>get_name()</td>
<td>get_char()</td>
</tr>
<tr>
<td>document</td>
<td>document</td>
<td>get_date()</td>
<td>get_char(),decode()</td>
</tr>
<tr>
<td>image</td>
<td>document</td>
<td>get_date()</td>
<td>decode()</td>
</tr>
<tr>
<td>text</td>
<td>document</td>
<td>get_date()</td>
<td>get_char()</td>
</tr>
<tr>
<td>image</td>
<td>image</td>
<td>decode()</td>
<td>get_char()</td>
</tr>
<tr>
<td>text</td>
<td>text</td>
<td>get_char()</td>
<td>decode()</td>
</tr>
</tbody>
</table>

Table 2 Illustrating constraints on coupling

“Type checking” in the manner just described has not been implemented in DEVS/HLA. We can foresee a
future implementation of such checking when reusable components become widespread and the attendant need to assure meaningful coupling of such components becomes critical.

Conclusions

We have reviewed the support of the DEVS formalism for hierarchical modular model construction and component reuse. We have shown that the basic concept of modularity for dynamic systems is more demanding than that of computational objects and is supported in the basic DEVS formalism, especially through the external transition function. We further showed that closure under coupling supports the ability to “componentize” a coupled model. This in turn, justifies and enables the application of, hierarchical model composition constructs. The underlying structure of hierarchical models is captured in the system entity structure (SES) which supports application to construction, management, and reuse of combinatorially-based families of models – as most large scale model development projects are likely to generate. We discuss the implementation the SES in DEVS/HLA with illustrations from examples developed in the DEVS/HLA (JAVA) environment.

References


BERNARD P. ZEIGLER is Professor of Electrical and Computer Engineering at the University of Arizona, Tucson. He has written several foundational books on modeling and simulation theory and methodology. He is currently leading a DARPA sponsored project on DEVS framework for HLA and predictive contracts. He is a Fellow of the IEEE.

HESSAM S. SARJOUGHIAN is Assistant Research Professor of Electrical and Computer Engineering at the University of Arizona, Tucson. His research areas of interest include theory and practices of distributed/collaborative modeling & simulation. Currently, he is leading the design and development of a collaborative modeling environment to support distributed simulation.