DEVS DIAGRAM REVISED: A STRUCTURED APPROACH FOR DEVS MODELING

Hae Sang Song
Dept. Computer Engineering
Seowon University
Cheongju, 361-742
Korea
hssong@seowon.ac.kr

Tag Gon Kim
Dept. Electrical Engineering
KAIST
Daejon, 305-701
Korea
tkim@ee.kaist.ac.kr

KEYWORDS
DEVS Diagram, Port, Variable, Phase Transition Diagram

ABSTRACT

Discrete Event Systems Specification (DEVS) formalism has been used in recent decades for modeling and simulation of discrete event systems as well as for some hybrid systems, since it has a sound semantics for modular hierarchical semantics and good simulation development environments. Nevertheless there are still deep gaps in understanding between the formalism and the practical modeling of large complex systems since the number of events and states of the systems are so huge to handle in the classical DEVS formalism. To solve this problem, this paper proposes the DEVS diagram, a structured diagramic form of the DEVS formalism. For this we introduce the essential notions of state variable and phase for structuring sequential states, and port and message for structuring sequential events. Phase transition diagram is also formally defined. We present the DEVS diagram in both graphical notations and mathematical formulations. A simple example illustrates the modeling method in the DEVS diagram.

1. INTRODUCTION

Demands for Modeling and Simulation (M&S) demands has grown in recent decades and Discrete Event Systems specification(DEVS) based simulation applications have become more prolific especially in the domain of large complex systems, for DEVS has a sound mathematical semantics for modular hierarchical semantics and has good simulation development environments (Kim 2010). Nevertheless a fresh M&S engineer still experiences difficulty in applying the classical DEVS formalism as is to a practical modeling of complex discrete event systems. This occurs for two reasons: 1) the formalism itself is too mathematical for practical use and the behavior is distributed into four separated functions; thus no integrated picture of a model is provided for the modeler to model and understand intuitively; and 2) in reality the number of sequential states or events is too large, because of the complexity of systems under consideration.

For these reasons we need a more structured form of sequential states and sequential events to deal with these complex systems in M&S. Among DEVS-based simulation development environment, DEVSim++ (Kim 1992) introduces the notions of messages and phases rather than events and sequential states. Nevertheless, the environment does neither fully implement these concepts nor explicitly fomalize this idea. Moreover, in the real-world fields, informal graphical modeling notations of their own have been utilized as an essential step before translating the graphical models into the mathematical DEVS model. There has been, however, no research on formally defining a graphical language conforming to the DEVS formalism in higher level modeling.

There have been a few efforts on how to represent DEVS models in more easier ways rather than in mathematical one. Most of them are either graphical approach or language based approach. The original version of DEVS diagram revised here is depicted at an example of RT-DEVS model (Hong and Song et al. 1997). The diagram notation has been used for a large number of commercial projects for large sized defense modeling simulation fields (Kim et al. 2010) with enriching the expressiveness. GGAD (Generic Graphical Advanced environment for DEVS Modeling and simulation) is a tool that adopts the diagram but with somewhat different appreance (Moallemi and Wainer 2010). Language approach for DEVS modeling is also presented by a work (Hong and Kim 2006). An extension of UML for DEVS is proposed as a SysML/DEVS profile (Nikolaidou et al. 2008). Also the DEVS standardization group has tried to make standards for the exchangibility of DEVS models. However, we think that DEVS has to have its original diagram for modeling standard, rather than depending on existing one like UML to best express the mathematical model. It should also supports necessary new notions for modeling large and complex systems, such as co-modeling (Kim 2006) and structuring states and events. Thus this paper is an effort to revise the original diagram(Hong and Song et al. 1997), as well as to support the mathematical foundation of the diagram.

Consequently this paper makes an effort to clearly define the graphical language for implementation models, called the DEVS diagram, and to narrow gaps of mathematical models in the classical DEVS formalism and implementation models in the graphical language. To handle large complex discrete event systems, the notion of port and message as a structured form of events, and that of state variable and phase for a structured form of sequential states are introduced. These efforts would make it easier for M&S engineers to model and implement large complex discrete event systems based on the DEVS formalism.

This paper is organized as follows. The next section introduces the classical DEVS modeling background. Then

we propose the definition of the DEVS diagram. A simple example illustrates the modeling approach in the course of definitions. Finally, we conclude the discussion.

2. BACKGROUND

2.1 The Classical DEVS Formalism

As it is well-known, the classical DEVS formalism can specify a system in two aspects: one for the behavior of a basic component, and the other for the overall structure of a system. An atomic DEVS formalism describes the behavior of a unit component not further decomposable, which consists of three sets and four functions.

$$\mathrm{AM} =< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$$

, where

X: input event set,

Y: output event set,

S: sequential state set, and total state set

$\mathrm{Q} = \{(\mathrm{s,e}) \mid \mathrm{s} \in \mathrm{S}, 0 \le \mathrm{e} \le \mathrm{ta(s)}\}$,

$\delta_{\mathrm{ext}}: \mathrm{Q} \to \mathrm{Q}$: external transition function, for $\delta_{\mathrm{ext}}(\mathrm{s,e,x}) = (\mathrm{s'}, \mathrm{e'}), \mathrm{e} < ta(\mathrm{s}), \mathrm{e'} = 0.$

$\delta_{\mathrm{int}}: \mathrm{Q} \times \mathrm{X} \to \mathrm{Q}$: internal transition function, for $\delta_{\mathrm{int}}(\mathrm{s,e}) = (\mathrm{s'}, \mathrm{e'}), \mathrm{e} = \mathrm{ta(s)}, \mathrm{e'} = 0.$

$\lambda: \mathrm{Q} \to \mathrm{Y}$: output function, for $(\mathrm{s,e}) \in \mathrm{Q}, \mathrm{e} = \mathrm{ta(s)}$

$\mathrm{ta}: \mathrm{S} \to \mathrm{R}_0^+$: time advance function,

R_0^+ is the non-negative real number set.

There are two types of transitions of a model: 1) external transitions entailed by external events; and 2) internal transitions in the case of no event occurrence until current state sojourn time has elapsed. In the latter case, just before the internal transition, an output event is producted at the state. In an analogy to the continuous systems, external transitions would correspond to the input driven state transition and internal ones the input-free state transition.

The coupled DEVS formalism specifies the structure of discrete event systems composed of components communicating with each other through event couplings,

$$\mathrm{DN} =< X, Y, M, EIC, EOC, IC, SELECT >$$

, where

X: input event set,

Y: output event set,

M: component model set, either atomic models or coupled models

$\mathrm{EIC} \subseteq \mathrm{DN.X} \times \bigcup_{\mathrm{i}} \mathrm{M_i.X_i}$: external input coupling relation,

$\mathrm{EOC} \subseteq \bigcup_{\mathrm{j}} \mathrm{M_j.Y_j} \times \mathrm{DN.Y}$: external output coupling relation

$\mathrm{IC} \subseteq \bigcup_{\mathrm{j}} \mathrm{M_j.Y_j} \times \bigcup_{\mathrm{i}} \mathrm{M_i.X_i}$: internal coupling relation

$\mathrm{SELECT}: 2^{\mathrm{M}} - \varnothing \longrightarrow \mathrm{M}$: select function

Notice that the coupled DEVS formalism above has the closure property, i.e., a coupled model may contain another coupled models as well as atomic models as its components. It captures the structure of a system, the components hierarchy and the interfaces between components. The SELECT function relates to the simultaneous scheduling problem of simulation that arranges the priorities of components when more than one component is to be scheduled at the same time.

2.2 DEVS Graph

For comprehensiveness, the behavior of an atomic model $\mathrm{M} =< X, \mathrm{Y}, \mathrm{S}, \delta_{ext}, \delta_{int}, \lambda, ta >$ can be depicted in a DEVS graph:

$$\mathrm{AG} =< \mathrm{N}, \mathrm{E} >$$

, where

$\mathrm{N} = \mathrm{S}$: the same as the sequential state set of M.

$\mathrm{E} \subseteq \mathrm{N} \times (\mathrm{X} \bigcup \mathrm{Y} \bigcup \{\varepsilon\}) \times \mathrm{R}_0^+ \times \mathrm{N}$, ε is the null event, where for an external transition $(s, x, e, s') \in \mathrm{E}, \in \mathrm{X}$ $\delta_{ext}(s, e, x) = s'$, and for an internal transition $(\mathrm{s, y, e, s'}) \in \mathrm{E}, y \in \mathrm{Y}, \delta_{int}(s, e) = s', \lambda(s) = y, e = ta(s)$.

Note that, by the definition of the atomic DEVS formalism, there is only one internal transition at a state $\mathrm{s} \in \mathrm{S}$. The null event is a pseudo-event representing no event occurrence at a time.

Figure **1** shows the notation in graphical form.

Figure **1** (a) depicts the atomic DEVS model behavior in graphical form; at the initial state (s_0,0), if there is no event (ε) until elapsed time e, then the total state becomes (s_0,e); then if there is still no event until (ta(s_0)-e) elapses from (s_0,e), an internal transition occurs to reach state (s_1,0); or if an input event x arrives at that total state (s_0,e), an external transition makes the state (s_2,0). For simplicity, this behavior can be depicted as in (b) and furthermore, it becomes a more compact form as in (c). Thus, the original DEVS graph can also be represented by $\mathrm{AG'} =< N', E' >$, where $\mathrm{N'} \subseteq \mathrm{S} \times \mathrm{R}_0^+$, where $\mathrm{N'} = \{(\mathrm{s,r}) | \forall \mathrm{s} \in |\mathrm{S}, \mathrm{r} = \mathrm{ta(s)}\}$, $|\mathrm{N'}| = |\mathrm{S}|$ and $\mathrm{E'} \subseteq \mathrm{N'} \times (\mathrm{X} \bigcup \mathrm{Y}) \times \mathrm{N'}$ as in

Figure **1** (c). If no time advance is specified at a state, it is assumed to be infinity (waiting forever at that state). Note that by definition, there is at least one internal transition defined at a node.

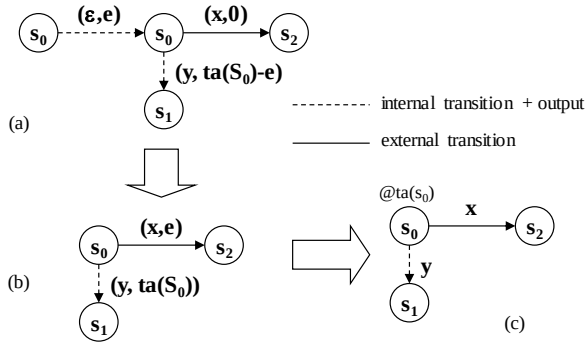

Figure 1. DEVS graph notation

Since both atomic and coupled models have their external interfaces set X and Y we represent a model *m* as a box B_m with event interfaces $B_m.I = \mathrm{X} \bigcup \mathrm{Y}$ on the border. We denote by $B_m.X$ the set of input event interfaces set of box B_m. Now a coupled model $\mathrm{N} =< \mathrm{X}, \mathrm{Y}, \mathrm{M}, EIC, EOC, IC, SELECT >$ can be represented in a graphic form in a straightforward manner; a coupled box model of model N is represented by $\mathrm{B_N} = <\mathrm{B_M}, \mathrm{C}>$ where $\mathrm{B_M}$ is the set of component box models $\mathrm{B_M} = \{\mathrm{B_{M_i}} | \mathrm{M_i} \in \mathrm{M}\}$, and $\mathrm{C} \subseteq \mathrm{B.I} \times \mathrm{B.I}, \mathrm{B} = \mathrm{B_M} \bigcup \{\mathrm{B_N}\}$ is the arc set each arc of which links an interface on a box to other interfaces as specified in N as EIC, EOC, and IC. Figure 2 is

an example of the box representation of a coupled model M12, where EIC={(M12.in, m1.x1)}, EOC = {(m2.out1,M12.out)}, and IC={(m1.out, m2.in), (m2.out2, m1.x2)}. Note that model m1 is in reality an instantiation of model M1 and m2 M2. The SELECT function will be described in a list of maps in text format {m1, m2,m3} → m1, elsewhere in the graph.

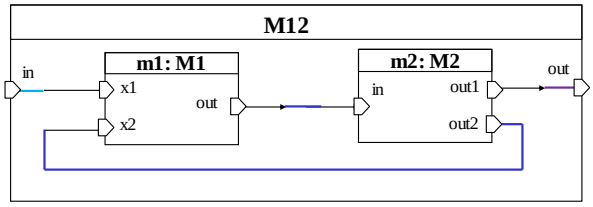

Figure 2. Coupled DEVS graph notation

2.3 Problem Statement

Although the DEVS formalism and the DEVS graph form specify discrete systems in a sound modular and hierarchal manner in system theoretical form and are adequate for logical analysis, in our experience, they too low level for the application of large and complex systems' modeling simulations. Thus a usually higher level of model representation has often been used in ad-hoc graphical forms in war game modeling simulations (Kim 2010). However, to author's best knowledge, there has been no explicit literature dealing with a higher level graphical representation of the DEVS formalism in structured form. Therefore this paper focuses on explicitly defining a higher level diagram, called the *DEVS diagram*, which is a structured form of DEVS graph presented earlier. For this purpose, we introduce the notion of port and message as a structured form of events, and that of state variable and phase for a structured form of sequential states to handle large complex discrete event systems. We, in this paper, focuse on presenting the DEVS diagram in graphical notations as well as mathematical forms.

3. STRUCTURING EVENTS AND STATES

The term 'structuing' means doublefold: one for state structuring and the other for event structuring. the event will be structured to 'ports' and the sequential states structured to 'variables' and 'phases'

3.1 Structuring Events: Port and Message

Descrete event systems interact with each other by sequential events in the classical DEVS formalism. In real-systems modeling, however, the number of events are so huge and sometimes infinite that we cannot treat it easily by directly applying the DEVS formalism in the course of modeling. Thus, one way of coping with this problem is to group these sequential events into related categories to handle it easier way.

In practical applications, there is a collection of an equivalent class of events, in the sense that these events influence from the same source to the same destination model, and the type of the events can be regarded as the same class or a group. We call an equivalent class of events a *message* and we can handle message flow via the notion of *port* and *channel*. A *port* is an arrival or departure place with a *port type* where thmessages of the same type can be handled. We denote the type of a variable or a port using operator $dom(\cdot)$. A *channel* is a connection from a port to another port; with channels, it is assumed a message at a departure port can be transferred instantly to connect arrival ports. Hereafter, we will denote the input port set by $\mathbf{P_X} = \{(\mathbf{p_x}, \mathbf{d_x})\}$, where $\mathbf{p_x}$ is a port and $\mathbf{d_x}$ is the domain of the port; through the port messages of the same domain (or port type) can be sent or received. Similary we denote by $\mathbf{P_Y} = \{(\mathbf{p_Y}, \mathbf{d_Y})\}$ the output port set. A port coupling between ports is called a channel and conversely a channel connects from a source port to the target port of the acceptable types.

The channel differ in that events are grouped into messages flowing through ports and channels. Note that an event is really an occurrence of sending or receiving a message of specific value though a channel. Since the channel transmission is instantaneous, an output event and the input event caused by a message transmission occur at the same time. In this sense a channel is a collection of event couplings with the same source and destination models compared to the classical DEVS formalism.

With the notions of port, message, and channel we can define a structured form of the classical DEVS formalism, which we call the *Structured DEVS Formalism*. We will not present it here due to the lack of space, which will be presented soon in an another paper. Note that it looks similar to the classical atomic DEVS formalism except that it is composed of structured states and events: port and message, and state variables, presented below.

3.2 Structuring States: State Variables and Phase

Analogy to the continuous systems specified in differential system equations, the concepts of system state variable can be applied to the discrete event systems especially in the DEVS formalism. A system state can be specified by a set of system variables, each of which has its domain. In general, a set of related states is usually grouped into a state variable; formally, a *state variable* is a container that can accommodate a group of related states which we call the domain of the variable. Thus a set of system sequential states can be structured to a set of system variables. Thus a system state is a combination of values that the state variables have at some time. This is a useful measure to form a structure of the flat sequential states.

Although state variable is one measure used to structure the sequential states, we need to further abstract them to a higher level of states, called a phase. Formally, a *phase* is a representative value of a set of equivalent states which produce the same output event and/or have the same time advance at the states. If we add phase variable(s) to the system state variables set, we can simplify the state transitions even more by fewer number of phase transitions. Now consider that we partition the composite state set V into equivalent classes such that each class has a set of sequential states with the same state sojourn time and/or output event. Let each class have a single representative name, called a phase, and we can add an additive or sometimes redundant phase variable $\boldsymbol{\psi}$ to the state variable set $\boldsymbol{S_V}$; it then becomes that $\boldsymbol{S'_V} = \boldsymbol{S_V} \cup \{\boldsymbol{\psi}\}$. We designate a phase variable as a high-level state variable, and the original ones as the low-level state variables. Then, a state of the system becomes a

combination of a phase value and a composite state of state variables, except for the phase variable. An extremely trivial redundant case occurs when a phase has only a state as its member, a one-to-one correspondence.

A phase can be hierarchical; that is, a phase can be decomposed more into *sub-phases* having disjoint composite state members, and so on. It is always true that a union of sub-phases within a phase gives the total states set of the phase and the intersection of all sub-phases results in an empty set, i.e., for a phase variable $\boldsymbol{\psi} \in \mathbf{S_V}$, $dom(\boldsymbol{\psi}) = \{\varphi_i\}$, let $V_{\varphi_i} \subseteq \mathbf{V}$ be the phase member states set of phase φ_i, then $\bigcup_i V_{\varphi_i} = \mathbf{V}$, and, for any two phases, it should be $\varphi_i, \varphi_j \in \boldsymbol{\psi}, V_{\varphi_i} \cap V_{\varphi_j} = \emptyset$. To discriminate phase member states, we define a *guard* on the composite state set $\mathbf{V}$ as a logical expression on low-level state variables that further filters a phase into a subset of states of the phase, or a sub-phase. For a guard G we denote by $V_G = \{v \mid G(v) = \text{true}, v \in \mathbf{V}\}$ the *guard member states set*. Then, for phase $\varphi \in \boldsymbol{\psi}$ and the phase member states set V_φ, if for a guard G, $V_G = V_\varphi$ we then call G the *phase guard* and denote it by G_φ. We also call a function $A: \mathbf{V} \to \mathbf{V}$ an (state transition) *action*. We can now define the notion of a phase transition diagram.

3.3 Phase Transition Diagram

Let $\mathbf{P_X}, \mathbf{P_Y}, \mathbf{S_V}$ be an input ports set, an output ports set and state variables set of a structured atomic model. Let $\boldsymbol{\psi} \in \mathbf{S_V}$ be a phase variable where each $\varphi_i \in \text{dom}(\boldsymbol{\psi})$ has a disjoint composite member states set $V_{\varphi_i} \subseteq V$ with its unique phase guard G_{φ_i}. We then formally define the specification of the *phase transition diagram* as the following.

$$PD = < N, E >$$

where

- $N = \{(\varphi, T_\varphi, V_\varphi) \mid \text{phase } \varphi \in dom(\boldsymbol{\psi}), \boldsymbol{\psi} \in \mathbf{S_V}\}$: a node set, where T_φ is the time advance of a phase φ, V_φ is the disjoint composite member states set that can be also described by a guard G_φ, $V_{G_\varphi} = V_\varphi$.
- $E \subseteq N \times (\{G_i\} \times \Sigma \times \{A_i\}) \times N$: a phase transitions set, $\{G_i\}$ is a sub-guards set, $\Sigma = \mathbf{X} \cup \mathbf{Y}$, the union of structured input / output event sets obtained from $\mathbf{P_X}, \mathbf{P_Y}$, respectively. $\{A_i\}$ is an actions set such that $A_i : \mathbf{V} \to \mathbf{V}$,

with three constraints for any two phase transitions on a phase,

1) (functionality) if the two events are the same, then the guards are disjointed with each other, i.e., $V_{G_i} \cap V_{G_j} = \emptyset$, and
2) (uniqueness) if the two events are different output events, then the guards are disjointed with each other, and
3) (integrity) the target state which is caused by the action of a phase transition is a member of the target phase members identified by the target phase guard for integrity.

A phase transition occurs by a **guard/event/action** pair. In essence a phase transition is a collection of sequential state transitions of equivalent sequential states. Formally the source states of two state transitions are said to be equivalent if the state sojourn time is the same; and the event; and destination states reached by the action are equal. Figure 3 illustrates the notion of phase transition; there are three phases $\varphi_1, \varphi_2, \varphi_3$ where φ_1 is further decomposed by guards on φ_1, G_1, G_2, G_3 such that $G_1(\varphi_1) = \varphi_{11}$, $G_2(\varphi_1) = \varphi_{12}$, $G_2(\varphi_1) = \varphi_{13}$. The states of sub-phase $\varphi_{11} = \{s_1, s_2, s_3\}$ that are all mapped to state s_9 of phase φ_2 by event $x_1?m_1$ are equivalent by definition. Conversely it can be regarded that we group these three sequential equivalent state transitions into a phase transition with a guard $G_1(\varphi_1) = \varphi_{11}$, an event $x_1?m_1$, and an action $A_1(\mathbf{S_V})=s_9 \in \varphi_2$, which is denoted by a phase transition $(\varphi_1, G_1, x_1?m_1, A_1, \varphi_2)$.

.

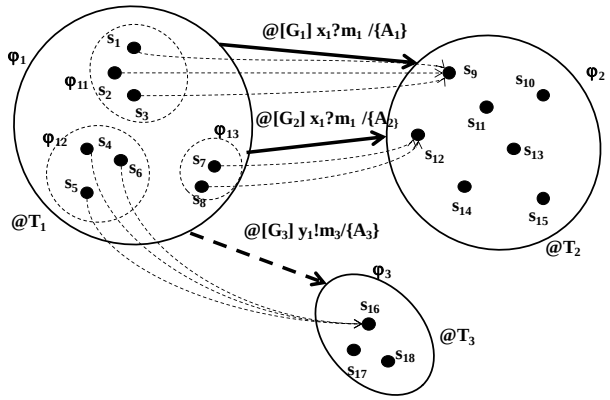

Figure 3 Illustration of the notion of phase transition

We define a graphical form of a phase transition diagram as follows. A phase is represented by a rounded box with its phase name and its time advance **@T** or multiple time advances with disjoint guards upon the phase, **{@[G_i]T_i}** below the name**.** An internal phase transition is drawn as a dotted arc from a source phase to a target phase along with an annotation, ***outport!message@[guard] / {action}***; this implies that when a composite state of the source phase meets the guard *guard* and the phase sojourn time elapses without any external event, we take the internal transition sending a message of value *message* to the output port *outport* and then change the state variable as specified by the action *action*. Likewise an external phase transition is described by a solid arc with the annotation ***inport?message @[guard] /{action}***; this also means that when a composite state of the source phase meets the guard *guard* and an input message arrived at the input port *inport* is the expected value specified by *message*, then take the external transition while executing the action *action*. If multiple messages cause the same action, it can then be specified by a message guard such as ***inport?[message guard] @[guard] /{action}***. If the message is not specified, it means a null message. If the guard is not specified, then it means the transition applies to all states of the source phase. If the action is not specified, it implies that only the phase variable changes are specified in the diagram. Finally we think that the phase guards are described implicitly or explicitly somewhere elsewhere in the diagram if a phase node has a small space.

Let us take a look at variables in more detail. Recall that a guard is a logical expression on state / phase variables which results in a subset of composite states that meet the guard. For practical usage, we can divide the state variables into two categories: those that affect state transition directly; and

the others that are irrelevant to any state transition in any way. The former are called the primary state variables which appear on the guard, and the latter user variables are those that any guard does not care about, though the user variables may be changed by the action. It is important to discriminate whether a state variable is a primary or a user variable in modeling an atomic model since the complexity of modeling can be reduced.

4. THE DEVS DIAGRAM

For engineers sometimes a graphical-language-based approach such as UML is preferred to the equivalent mathematical one, especially when modeling and design stage in the system development. it is because that we can easily capture the intuitive picture of a system under consideration, not yet cleared. As indicated in the DEVS graph for the classical DEVS, is a too a low level to take a picture of a large complex system. The DEVS diagram proposed and refined here is another instrument for modeling a system in the structured form. The notion of the phase described above is an essential tool for structuring and abstracting state transitions while that of the state variable is essential for sequential states and that of the port for sequential discrete events. As described earlier a phase variable containes a set of phase values, each of which represents a subset of composite states. We can now define the DEVS diagram formally.

4.1 Atomic DEVS Diagram

The DEVS diagram consists of descriptions of an atomic DEVS diagram and a coupled DEVS diagram. An atomic DEVS model can be specified as a *atomic model box* with input and output ports on the border and the model name at the top; a variable box inside the model box has a list of state variables with their initial values, and most importantly a phase transition diagram. Formally, an atomic DEVS diagram of atomic model M is specified by

$$\mathrm{ADD_M} =< \mathrm{AB_M}, \mathrm{P_M} >$$

, where

$\mathrm{AB_M} =< \mathbf{P_X}, \mathbf{P_Y}, \{\mathbf{v}: \mathbf{d_v} = v_0\} >$: an atomic model box of name M, with ports on the box boundary, where an input(output) port is depicted by an inward(outward) box-arrow annotated by ***port:type***, $\mathbf{p_x}: \mathbf{d_x} \in \mathbf{P_X}, (\mathbf{p_y}: \mathbf{d_y} \in \mathbf{P_Y})$, respectively. A variable box inside the model box lists the variables, each variable of which is described by $\mathbf{v}: \mathbf{d_v} = v_0$, variable name $\mathbf{v}$, domain of the variable $\mathbf{d_v}$ and an initial value v_0.

$\mathrm{P_M} =< N, E >$: a phase transition diagram inside the model box.

Figure 4 shows an example of an atomic DEVS diagram whose model name is 'ABuffer'; it has two input ports named 'in' with type 'Job' and 'done' and one output port 'out' with type 'Job'. In the variable box there is a phase variable usually named 'phase' and two primary state variables, 'p' of type {B,F} and 'n' of non-zero integer. There is one internal phase transition from phase SEND to WAIT by 'out!w/{n--,p=B}', which means there is no guard, and sends a message 'w' to the output port 'out'; then change the state variables 'n' and 'p'. The default initial value of variable 'phase' is WAIT which is also bolded in the phase transition diagram. The initial values of the variable box can be changed at a coupled model specification if required. By convention prefix 'A' of a model name means an atomic model and 'C' a coupled model.

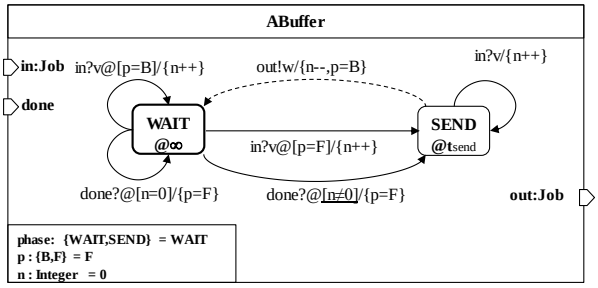

Figure 4. Example of an atomic DEVS diagram

In more detail the above phase transition diagram is an example of a case that can be derived from a state transition graph in **Figure 5**. For two state variables 'p' and 'n', *type(*p)={B,F} and *type(*n) = {0,1,2,..}, the composite states in the Figure are partitioned into two phases: WAIT={(n,p)|(n=0 and p=F) or p = B} and SEND = {(n,p)| n > 0 and p = F}. The time advance or state sojourn time at phase WAIT is infinite and that of SEND is t_{send}. At phase WAIT if a message 'v' arrives at port 'in' at a sub-phase by identified guard @[p=B] (left top) then it returns to phase WAIT taking action {n++} or for the same event at a sub-phase @[p=F] (center), then it goes to phase SEND with doing action {n++}. The phase variable 'phase' is implicitly added to the system variables and the change of the phase variable is implicitly assumed. Conversely we can extract a state transition graph from a phase transition diagram. Note that the phase transition diagram above is much simpler than the state transition graph below; this is the reason why we prefer the phase transition diagram to the state transition diagram of the DEVS graph.

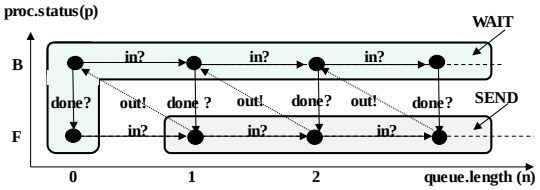

Figure 5. Mapping from a state transition graph

We remark that, although there is usually one phase diagram for an atomic model, there may be two or more phase transition diagrams in special cases in which the state variables can be grouped into two or more and the groups are independent of each other. By the term 'independent', we mean any composite state of a group will never affect any state transition of the other group. A phase may also have the closure property; that is, a phase can have a phase diagram in it with its sub-phases. The former case is a kind of vertical partition of state variables and the latter is a horizontal partition of a phase. For example, in **Figure 5**, the phase WAIT can be divided further into BUSY_WAIT = WAIT ∩ {(n,p)|p=B} and FREE_WAIT = WAIT ∩ {(n,p)|p=F} = {(0,F)}. Then some of the guards in **Figure 4** become unnecessary; so we can obtain a revised phase diagram, as in **Figure 6**.

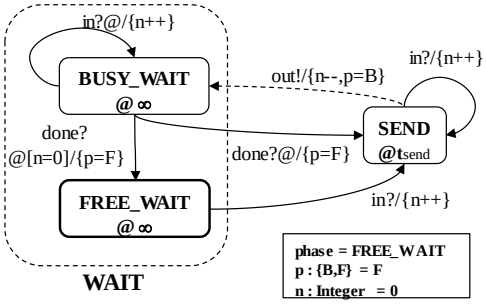

Figure 6 Phase partitioning

An example of variable partitioning is illustrated in **Figure 7**. To illustrate this notion, add a counter variable 'c,' accumulating up the number of inputs from port 'in'. We can add this variable and a new phase diagram with a new phase variable 'phase2,' while the rest of the model is the same as before, for the counter variable does not affect the existing phase diagram at all. This parallel modeling has advantages in its simplicity and comprehensiveness compared to an equivalent composite phase diagram. However we need take care of the time advance in the implementation stage. The time advance of the parallel phase diagrams should be the minimum of the remained time advances of the current phases, which should be managed by the developer in the time advance function. We recommend the developer that it is easy this model be implemented in two atomic models with the same interface, for the time management is delegated to the simulation engine.

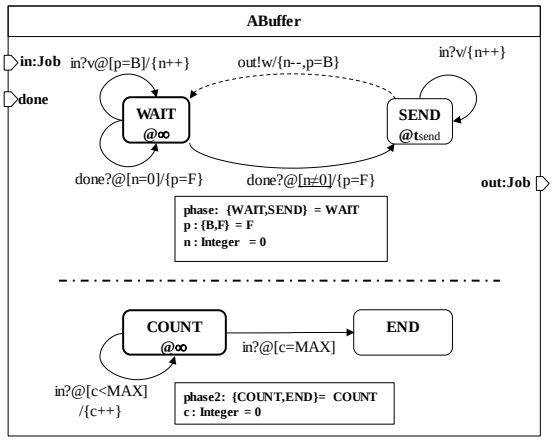

Figure 7. Parallel phase diagrams

On the other hand the methodology of collaborative modeling or co-modeling has had a good impact on multi-party M&S projects. The DEVS diagram supports this methodology in a simple way: 1) replace a state variable with an object, and 2) replace the action related to the state variable with the methods of the object. Recalling the **Figure 4** of the atomic buffer model, we can know that the state variable 'n' needs to be more refined, and we design a corresponding co-object 'q' of type 'Queue' instead of queue length 'n'. Related actions of the variable 'n' are either increasd (refined to inserte) or decreased (refined to delete); and, a required guard is to quest the length to the co-object 'q'. Furthermore we can add the input message type 'Job' for the port 'in'. Then we can obtain a refined and detailed atomic model based on the co-modeling methodology as shown in Figure 8. Action 'n++' is replaced by q.insert() , where the object 'q' will increase the queue length. The queue length can be queried by q.length() operation. Note that the input event is denoted by 'in?v' where 'v' is a temporary variable storing the job from input port 'in'. In this way, we can gradually refine a very abstract atomic model with only state variables first to a detailed model with corresponding co-objects. **Figure 9** is a sketch of simulation software for the model specification in Figure 8.

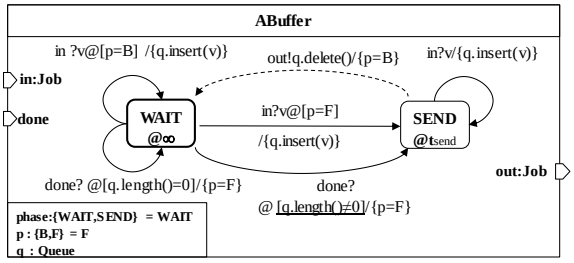

Figure 8 Refining with co-modeling methodology

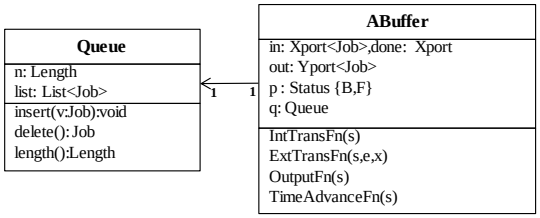

Figure 9 Sample software design of the buffer model

4.2 Coupled DEVS diagram

Formally, a coupled DEVS model N is specified by a coupled DEVS diagram,

$$\mathrm{CDD_N} =< \mathrm{B_N}, \{\mathrm{B_{m:M}}\}, \mathrm{C_N} >$$

where,

$\mathrm{B_N} =< \mathbf{P_X}, \mathbf{P_Y} >$: a outmost coupled model box of name M with ports on its boundary, where an input(output) port is depicted by an inward(outward) box-arrow annotated by ***port:type***, $\mathbf{p_x}: \mathbf{d_x} \in \mathbf{P_X}, (\mathbf{p_y}: \mathbf{d_y} \in \mathbf{P_Y})$, respectively.

$\{\mathrm{B_{m_i:M_i}}\}$: a set of model object boxes located inside the coupled model box $\mathrm{B_N}$, where a model object is denoted by $\mathrm{m_i}: \mathrm{M_i}$, a combination of object name '$\mathrm{m_i}$' of model '$\mathrm{M_i}$'.

$\mathrm{C_N} =< \mathrm{C_{EIC}}, \mathrm{C_{EOC}}, \mathrm{C_{IC}} >$: the channel specification, where a solid line connects each port pair in the sets,

$\mathrm{C_{EIC}} \subseteq \{(\mathbf{p_x}, \mathbf{p}'_{\mathbf{x}}) | \mathbf{p_x} \in \mathrm{B_N}.\mathbf{P_X}, \mathbf{p}'_{\mathbf{x}} \in \mathrm{B_{m_i:M_i}}.\mathbf{P_X}$ for any component model object $\mathrm{m_i}: \mathrm{M_i}\}$,

$\mathrm{C_{EOC}} \subseteq \{(\mathbf{p_Y}, \mathbf{p'_Y}) | \mathbf{P'_Y} \in \mathrm{B_N}.\mathbf{P_Y}, \mathbf{p_Y} \in \mathrm{B_{m_i:M_i}}.\mathbf{P_Y}$ for any component model object $\mathrm{m_i}: \mathrm{M_i}\}$,

$\mathrm{C_{IC}} \subseteq \{(\mathbf{p_Y}, \mathbf{p_x}) | \mathbf{p_Y} \in \mathrm{B_{m_i:M_i}}.\mathbf{P_Y}, \mathbf{p_X} \in \mathrm{B_{m_j:M_j}}.\mathbf{P_X}$, for any i, j$\}$

The couple model box $\mathrm{B_N}$ is similar to the atomic DEVS model box except for the variable box omitting. Component object boxes are to be placed inside the coupled model box. Recall that a model object differs from a model for it is an instantiation of a model. A model object box is the same as the model box except for the box name of the form of ***object:Model*** which represents a model object '***object'*** , an instantiation of the model '***Model***.' A channel is drawn with a solid line from a source port to the destination port.

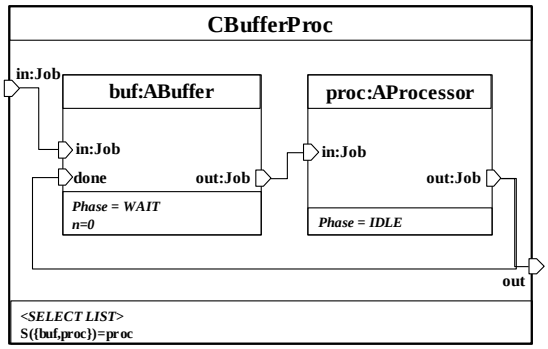

Figure 10. Example of coupled DEVS diagram

Figure 10 illustrates the notation by a coupled model named 'CBufferProc'. The type of port 'in' of the coupled model is 'Job'. There are two component objects: 'buf' of model 'ABuffer' and object 'proc' of model 'AProcessor'. There are four channels: (CBufferProc.in, buf.in), (buf.out,proc.in), (proc.out,buf.done), and (proc.out,CBufferProc.out). The SELECT function is described as a list at the lowest box.

5. CASE STUDY: A GBP MODEL

Using the DEVS diagram defined above we model a simple but full simulation model called GBP (generator-buffer-processor) model, a kind of queuing model such as bank teller model. Figure 11 depicts the whole system model 'CGBPSim', whose component models are two atomic models, 'gen:AGenerator' and 'trn:ATransducer' and a coupled model, 'bp;CBufferProc' as shown in Figure 10. A generator model object 'gen' creates jobs periodically; the transducer object 'trn' manages the statistics of the simulation result by accepting completed jobs from BufferProc model object 'bp'; and, BufferProc 'bp' is a coupled model object again composed of 'buf:ABuffer' and 'proc:AProcessor'; The model 'bp' is buffering and processing the job and outputs the completed job to the output port 'out:Job'. SELECT list is not depicted here but we assume that the priority is transitively 'trn' > 'bp' > 'gen'. In summary, a job message created by the generator model goes through the buffer-proc coupled model and then finally reached to the transducer. When the transducer determines to stop simulation, it sends 'STOP' command to the generator. The ports are connected from an output port to input ports according to the model specification. Note that 'CGBPSim' is not an object but a model.

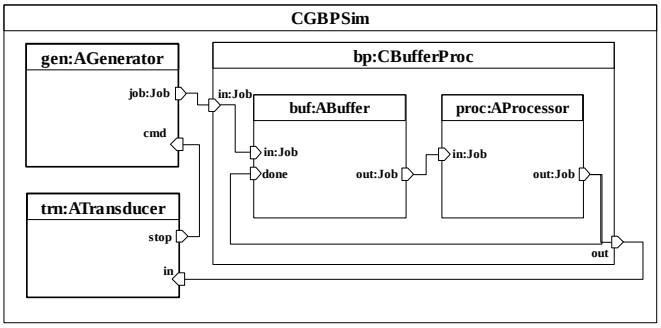

Figure 11. GBP model in coupled DEVS diagram

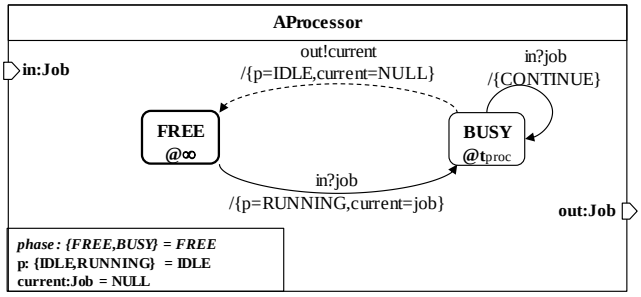

Figure 12 AProcessor: atomic processor model

Figure 12 is an atomic model of the processor. It has variables of the processor status 'p: {IDLE,RUNNING}', a temporary variables storing a job message currently processed, and the phase variable with trivial two phases: FREE={(p,-)| p=IDLE}, BUSY={(p,-)|p=RUNNING}.

The generator model in atomic DEVS diagram is shown in Figure 13. It receives commands via port 'cmd' whose type is {STOP,RESUME}. The main activity of the model is to create jobs periodically. We can observe the creation activity is delegated to a co-object 'factory:JobFactory', whose operation 'factory. create()' is periodically called and the resultant job is sent to the output port 'job:Job'. At phase 'CREATE' if it receives a command through temporary variable 'v' and the value of the variable is STOP, then the model goes to the phase 'STOP';

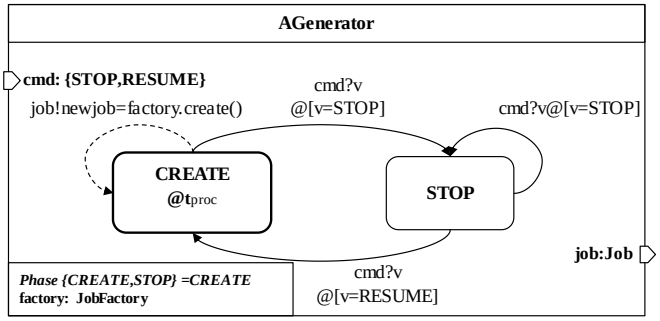

Figure 13 AGenerator: atomic generator model

Finally the transducer model shown in Figure 14 collects the completed jobs until the number reaches to a prespecified maximum, and then it sends a 'STOP' command.

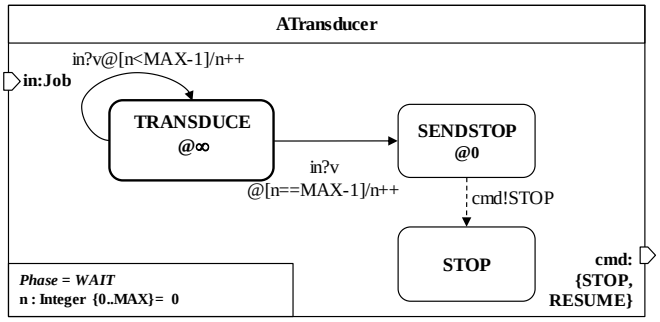

Figure 14 ATransducer: atomic transducer model

7. CONCLUSIONS

This paper proposed the DEVS diagram for practical modeling of large complex discrete event systems. The main idea was to introduce the notions of port and message, and

state variable and phase, which group sequential states and events into ports and variables. The diagram itself has been used for years but not defined formally and lacked a mathematical foundation; thus, sometimes it has lacked semantics, been incomplete, even violated the DEVS formalism. We did comprehensive and through work for the definition of the DEVS diagram to provide engineers with an efficient tool for modeling and design of simulation software. For lack of space, this paper can not, however, provide the through mathematical foundation based on the structured DEVS formalism; thus we made focus on the diagram itself. In the subsequence full paper to be submitted will prove that the DEVS diagram is based on the through mathematical foundation. Based on this work, we will update DEVS Specification Language in script form and plan to make a computer-aided modeling tool for DEVS based simulation development. More work, however, is required to implement DEVS simulation tool conforming the proposed DEVS diagram. As we expect DEVS applications to become more prolific as M&S demands grow, our work will be utilized more in the domain of complex M&S.

REFERENCES

Hong, Jun S.; Hae S. Song; Tag G. Kim; and K.H. Park. 1997. "A Real-time Discrete Event System Specification Formalism for Seamless Real-time Software Development," *Discrete Event Dynamic Systems*, Vol. 7, No. 4, pp. 355 – 375.

Hong, Ki J. and Tag G. Kim. 2006. "DEVSpecL-DEVS specification language for modeling, simulation and analysis of discrete event systems," *Information and Software Technology*, Vol. 48, No. 4, pp. 221 – 234.

Kim, Jae H. and Tag G. Kim. 2006. "Parametric Behavior Modeling Framework for War Game Models Development Using OO Co-Modeling Methodology," *2006 Sping Simulation MultiConf*., Huntsville, USA, pp. 69 – 75.

Kim, Tag G.; C. H. Sung; S.Y. Hong; J.H. Hong; C.B. Choi, J.H. Kim; K.M. Seo; and J.W. Bae. 2010. "DEVSim++ Tools Set for Defense M&S and Interoperation," *Journal of Defense Modeling and Simulation*, Sumitted.

Kim, Tag G. and S. B. Park. 1992. "The DEVS formalism: hierarchical modular systems specification in C++," 1992 European Simulation MUlticonference, York, United Kingdom. Pp. 152-156.

Moallemi, M. and Gabriel A. Wainer. 2010. "Designing and Interface for Real-Time and Embedded DEVS," *Proceedings of 2010 Spring Simulation Conference*, pp.154-161.

Nikolaidou, M.; V. Dalakas; L. Mitsi; G.D. Kapos; and D. Anagnostopoulos. 2008. "A SysML Profile for Classical DEVS Simulators," *Proceedings of 3rd Internal Conference on Software Engineering Advances, pp. 445-450.*

Song, Hae S. and Tag G. Kim. 2005. "Application of Real-time DEVS to Analysis of Safety-critical Embedded Control Systems: Railroad-crossing Control Example," *Simulation*, Vol. 81, No. 2, pp. 119 - 136.

Zeigler, B. P. and Tag G. Kim. 2000. *Theory of Modelling and Simulation (2nd Ed.)*, Acedemic Press.

BIOGRAPHY

Hae Sang Song was born in Damyang, Korea, and studied his MS.D and Ph.D courses in Electrical Engineering in KAIST (Korea Advanced Institute of Science and Technology) . He worked for a couple of years in an R&D lab, IAE(Institued of Advanced Engineering) in 1999-2000. He also worked in a venture company for about two years, and has been a professor of Dept. Computer Engineering of Seowon Univisity, Korea, since 2002. He spends his sabbatical year 2010 in Systems Modeling Simulation Lab of the co-author Tag Gon Kim, KAIST, as a visiting schalor for defense M&S projects. His major interest resides in modeling simulation, analysis, and control of discrete event dynamic systems.