The DEVS Formalism

Rhys Goldstein[1], Gabriel A. Wainer[2], Azam Khan[1]

[1]Autodesk Research
210 King St. East
Toronto, ON, Canada

[2]Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada

Abstract

The DEVS formalism is a set of conventions introduced in 1976 for the specification of discrete event simulation models. This chapter explains the core concepts of DEVS by applying the formalism to a single ongoing example. First, the example is introduced as a set of informal requirements from which a formal specification is to be developed. Readers are then presented with alternative sets of modeling conventions which, lacking the DEVS formalism's approach to representing state, prove inadequate for the example. The chapter exploits the DEVS formalism's support for modular model design, as the system in the example is specified first in parts and later as a combination of those parts. The concept of legitimacy is demonstrated on various model specifications, and the relationship between DEVS and both object-oriented programming and parallel computing is discussed.

Keywords: DEVS, discrete event simulation, modular model design, legitimacy, object-oriented programming, parallel computing

Introduction

The DEVS (Discrete Event System Specification) formalism is a set of conventions for specifying discrete event simulation models. It was introduced in 1976 with the publication of Bernard Zeigler's *Theory of Modeling and Simulation* (Zeigler, 1976). While the latest edition of that book (Zeigler et al, 2000) provides a comprehensive overview of DEVS theory, here we focus on the application of the core concepts. The chapter is organized around a particular example: the simulation of an automatic lighting system in an office environment. We develop this example from a set of informal requirements to a complete formal specification.

Before we begin, let us clarify the difference between a discrete time simulation and a discrete event simulation. Numerous simulations are implemented with a time variable t that starts at some initial value t_0, and increases by a fixed time step Δt between calculations. The flowchart in Figure 1 outlines the procedure.

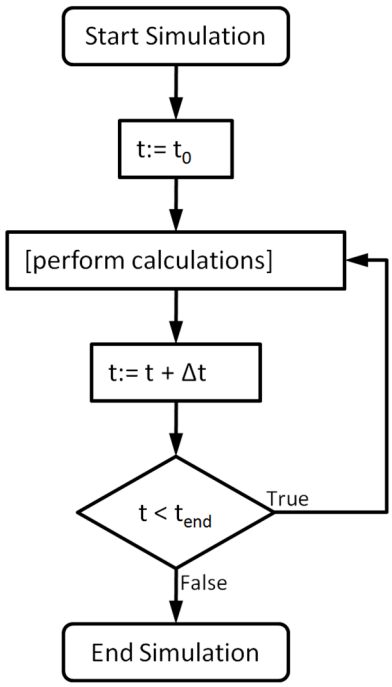

Figure 1. Discrete Time Simulation Procedure

This type of simulation is a ***discrete time simulation***, as *t* is effectively a discrete variable. The approach is simple and familiar, but limited in that the duration between any pair of inputs, outputs, or state transitions must be a multiple of *Δt*.

DEVS can be applied to discrete time simulation, but it is best suited to the discrete event approach for which it was invented. In a ***discrete event simulation***, time is continuous. Any pair of events can be separated by any length of time, and there is generally no need for a global *Δt*. Later in the chapter we will present a procedure like that in Figure 1, but suitable for all discrete event simulations.

The adoption of a discrete event approach impacts the model development process. For example, suppose one designs separate models for different parts of a larger system. Ideally, modeling the overall system would be a simple matter of combining these submodels. With discrete time simulation, one would have to choose a single *Δt* appropriate for every submodel, or invent some scheme by which only certain submodels experience events at any given iteration. With DEVS, two models can be coupled together regardless of how they handle time advancement. The only requirement is that the output values of one model are consistent with the input values of the other.

In this chapter we exploit the DEVS formalism's support for modular model design. First we present an example of a system as a combination of three interacting subsystems. We later specify an ***atomic model***, an indivisible DEVS model, for each of these subsystems. From there we specify a ***coupled model***, a DEVS model composed of other DEVS models, by combining the three atomic models. As we proceed from atomic models to coupled models, we analyze the legitimacy of various specifications. Towards the end of the chapter, we discuss DEVS in the context of object-oriented programming and parallel computing.

An Example

Buildings are believed to be responsible for roughly one third of greenhouse gas emissions worldwide (United Nations Environment Programme, 2009). Understandably, there is a growing interest in technologies that reduce the energy required for building operation.

Consider an automatic lighting system in an office building. At its simplest, such a system consists of a motion detector controlling a lighting fixture for a single workstation. When an office worker is present, the motion detector signals the lighting fixture, and the lighting fixture keeps the workstation illuminated. When the worker leaves, the motion detector signals the lighting fixture again, and the lights may turn off to save energy. We are interested in modeling the overall system; not just the Detector and the Fixture subsystems that compose the Automatic Lighting System, but the Worker as well. Combined, the three subsystems compose the Automatic Lighting Environment as illustrated in Figure 2.

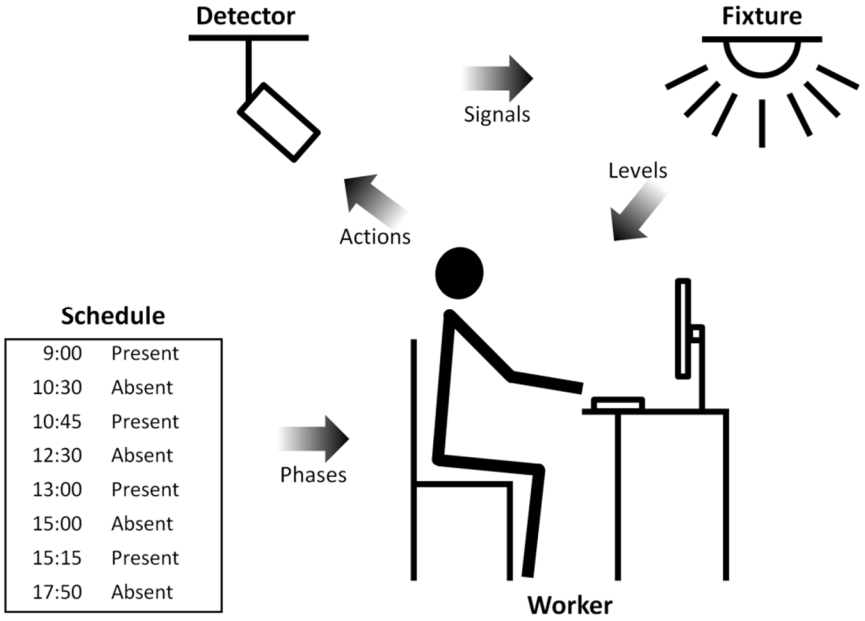

Figure 2. Conceptual Model of an Automatic Lighting Environment

Figure 2 identifies the type of information that can be transmitted to and from each subsystem. This transmitted information is organized into the four sets of values below. We can see, for example, that the Detector may send one of two values to the Fixture: “Still” or “Moving”.

$$Phases = \{\text{"Absent"}, \text{"Present"}\}$$
$$Actions = \{\text{"Gone"}, \text{"Typing"}, \text{"Reading"}, \text{"Waving"}\}$$
$$Signals = \{\text{"Still"}, \text{"Moving"}\}$$
$$Levels = \{\text{"Dark"}, \text{"Light"}\}$$

The specification we seek must accommodate arbitrary schedules, which indicate whether the Worker is present or absent at any given time. For now, consider a scenario resulting from the

specific schedule in Figure 2. At the beginning of the day, the Worker is absent and the light is off. At 9:00 the Worker arrives (*phase* = "Present") and begins typing on the computer (*action* = "Typing"). The Detector observes motion and informs the Fixture (*signal* = "Moving"). The Fixture responds by turning on (*level* = "Light"), and the Worker simply continues typing.

At 10:30, the Worker is scheduled to leave for a break (*phase* = "Absent"). The Detector will notice that the Worker has left (*action* = "Gone"), and signal the Fixture (*signal* = "Still"). The lights will then turn off (*level* = "Dark"), but not right away. The system is programmed such that there is a delay of Δt_{Saving} between the time when motion is last detected and the time when the lights are turned off to save energy. If Δt_{Saving} exceeds 15 minutes, the workstation will remain lit throughout the Worker's 10:30-10:45 break. Otherwise the Fixture will turn off after Δt_{Saving} elapses, then turn back on again when the Worker returns at 10:45.

Let us revisit 9:00, or shortly thereafter, when the lights are on and the Worker is typing. The Worker is to continue typing for a time of Δt_{Typing}, after which they switch to reading (*action* = "Reading"). They continue reading for a time of $\Delta t_{Reading}$, and then revert to typing for Δt_{Typing}, and then start reading again for $\Delta t_{Reading}$, etc. Relying on motion, the Detector is unable to distinguish between reading and absence. It simply informs the Fixture that there is no motion. Consequently, if Δt_{Saving} is less than $\Delta t_{Reading}$, the Worker may still be present when the Fixture turns off. If that happens, the Worker must make a waving gesture to trigger the Detector (*action* = "Waving"). The Worker resumes reading when the light turns back on.

An engineer could use simulation to help optimize automatic lighting systems like the one described. Their goal could be to determine an appropriate value for Δt_{Saving}. If Δt_{Saving} is too large, the technology would fail to save much energy when office workers are absent. But if Δt_{Saving} is too small, office workers would become annoyed for frequently having to "wave" the lights back on while reading.

Our goal is to provide a formal specification for a model of the overall system. This Environment model is a function of the three parameters described above.

$$Environment(\Delta t_{Saving}, \Delta t_{Typing}, \Delta t_{Reading}) = [\text{specification}]$$

We will begin by seeing what makes DEVS appropriate for the task.

Representation of State

One aspect of DEVS that sets it apart from other modeling formalisms is its approach to representing state. To understand the impact of how state is represented, let us first consider a formalism that neglects state completely. We will apply this formalism to the Detector, which is the simplest of the three subsystems described in the previous section. Recall that the Detector receives actions as input and sends signals as output. Hence, our formalism will allow us to define inputs, outputs, and a function that maps the former to the latter. Here is such a formalism, which we call Formalism A:

$\langle X, Y, \lambda \rangle$ is the structure of a Formalism A model specification

X is the set of input values

Y is the set of output values

$\lambda: X \to Y$ is the output function

Here is a Detector model specified using Formalism A:

$$Detector_A = \langle X, Y, \lambda \rangle$$

$$X = \{(\text{"action}_{\text{in}}\text{"}, action) | action \in Actions\}$$

$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) | signal \in Signals\}$$

$$\lambda\big((\text{"action}_{\text{in}}\text{"}, action)\big) = (\text{"signal}_{\text{out}}\text{"}, signal)$$

$$(action \in \{\text{"Typing", "Waving"}\}) \implies (signal = \text{"Moving"})$$

$$(action \in \{\text{"Gone", "Reading"}\}) \implies (signal = \text{"Still"})$$

There are a couple of things in the specification worth noting. First, we have defined an input port "$\text{action}_{\text{in}}$" and an output port "$\text{signal}_{\text{out}}$". A ***port*** is a label used to distinguish a particular type of input or output from other types of inputs or outputs. Ports are not strictly necessary for such a simple model, but we will make a habit of using them to help us combine models later on. All inputs and outputs will be defined as (*port*, *value*) pairs, as done above. Also note the two implications that define the output function. One maps both the "Typing" and "Waving" input actions to the output signal "Moving", and the other maps both "Gone" and "Reading" to "Still".

There is a subtle problem with this specification. If a "Reading" action is received, followed by "Gone", the model will output two consecutive "Still" signals. Or if it receives two consecutive "Typing" actions, it will send two consecutive "Moving" signals. We want the signals "Moving" and "Still" to be output in alternation only. If an input action would produce the same signal as its predecessor, the redundant output should be skipped. This implies that each output will depend not only on the current input, but on previous inputs as well. In other words, the model must have ***state***.

State is generally represented as a group of ***state variables***. State variables are analogous to model parameters in that they are associated with a single model and can affect that model's output. The difference is that ***model parameters***, like Δt_{Saving}, Δt_{Typing}, and $\Delta t_{Reading}$ in our automatic lighting example, remain constant throughout a simulation. State variables may be reassigned.

If we want to simulate the Detector model with Formalism A, we must define the simulation procedure associated with the formalism. Illustrated in Figure 3, the procedure is simple. As always, the time t starts at some initial time t_0. It then advances repeatedly to the time of the next event, which in this formalism is the time of the next input x ($x \in X$). At each event, the output function λ is evaluated to obtain the corresponding output y ($y \in Y$). Note that when the inputs are exhausted, we assume that "[time of next input]" is ∞.

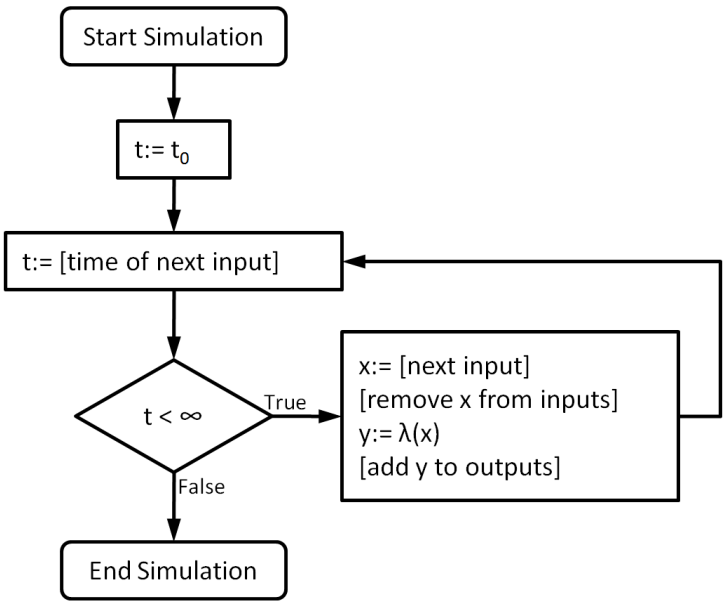

Figure 3. Formalism A Simulation Procedure

As we just discussed, models specified in Formalism A have no state (as the formalism does not have a representation of state). Formalism A models are therefore ***memoryless***. Among other things, this prevents us from avoiding identical consecutive output values (as in our Detector model). To address this issue in a more elegant fashion, let us propose a more complex formalism, called Formalism B.

Formalism B is similar to Formalism A in that events coincide with inputs. However, models now have state. The state of a Formalism B model remains constant between events, but may change during any event. Here is Formalism B:

$\langle X, Y, S, \delta, \lambda \rangle$ is the structure of a Formalism B model specification

X is the set of input values

Y is the set of output values

S is the set of states

$\delta: S \times X \to S$ is the transition function

$\lambda: S \times X \to Y \cup \{\emptyset\}$ is the output function

There are four differences between this formalism and the previous. First, a set of states S has been added. At any point, a model's state s must satisfy $s \in S$. Second, there is now a transition function δ that can change the model's state. Third, the output function λ now takes s as one of its arguments. Fourth, λ may result in $\emptyset$, indicating that the output is to be ignored.

By giving up Formalism A for Formalism B, we have accepted additional complexity for improved generality. The Detector model specification below is lengthier than the previous, but we have introduced behavior that we could not previously describe. The model has one state variable,

signal, which allows us to now check whether a newly received action would produce the same signal as the preceding action. If so, we now output ∅ instead of sending redundant information.

Using Formalism B, the Detector specification is as follows:

$$Detector_B = \langle X, Y, S, \delta, \lambda \rangle$$

$$X = \{(\text{"action}_{\text{in}}\text{"}, action) | action \in Actions\}$$

$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) | signal \in Signals\}$$

$$S = Signals$$

$$\delta\big(signal, (\text{"action}_{\text{in}}\text{"}, action)\big) = signal'$$

$$action \in \{\text{"Typing", "Waving"}\} \implies (signal' = \text{"Moving"})$$

$$action \in \{\text{"Gone", "Reading"}\} \implies (signal' = \text{"Still"})$$

$$\lambda\big(signal, (\text{"action}_{\text{in}}\text{"}, action)\big) = (\text{"signal}_{\text{out}}\text{"}, signal')$$

$$\begin{pmatrix} action \in \{\text{"Typing", "Waving"}\} \\ signal = \text{"Still"} \end{pmatrix} \implies (signal' = \text{"Moving"})$$

$$\begin{pmatrix} action \in \{\text{"Gone", "Reading"}\} \\ signal = \text{"Moving"} \end{pmatrix} \implies (signal' = \text{"Still"})$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \implies (y = \emptyset)$$

Observe that the transition function δ records the previous output, either “Moving” or “Still”, in the state variable *signal*. Likewise, note the changes to the output function λ. The two implications are still there, but now the conditions depend on *signal*. We only output “Moving” if *signal* was previously “Still”, and we only output “Still” is *signal* was previously “Moving”. There is also a third implication: if the neither of the first two conditions are met, we output ∅.

Figure 4 shows the simulation procedure associated with Formalism B. Note the inclusion of *s*, its initial value s_0, its reassignment using δ, and the changes to λ.

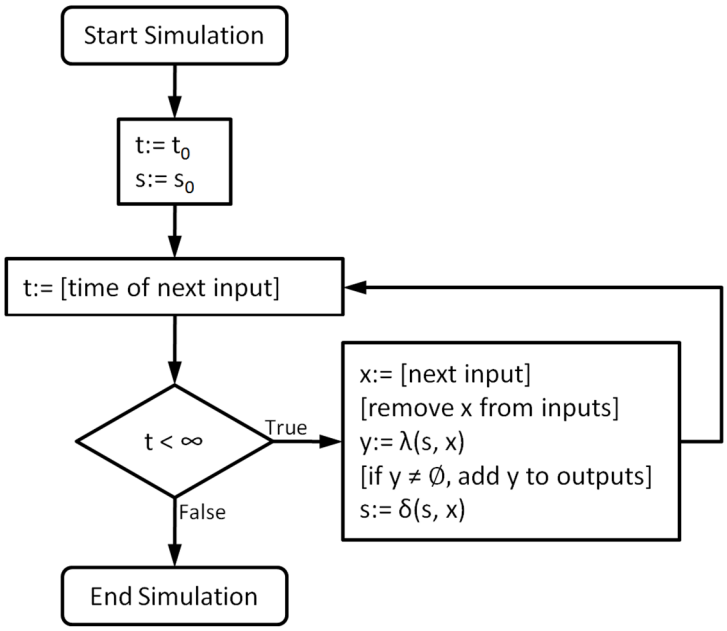

Figure 4. Formalism B Simulation Procedure

Formalism B appears well suited to the Detector model. However, our modeling requirements are about to get steeper. Consider the Fixture model. After receiving an input signal of "Still", the lights will turn off after a time of Δt_{Saving}. So after Δt_{Saving} elapses, the Fixture model must spontaneously send an output without having received an input at the same time. Such internally triggered outputs are not possible with Formalism B. But Formalism B and others like it have an even more fundamental problem. The problem pertains to how state is represented.

In Formalism B, state remains constant between events. The problem is that the state of a real-world system may change continuously over time. Take the automatic lighting system, for example. After the office worker leaves their workstation, the lighting system is in such a state that it will turn off after a time of Δt_{Saving}. One infinitesimal duration dt later, the lighting system is in an entirely new state: a state in which it will turn off after a time of $\Delta t_{Saving} - dt$. The system passes through an infinite number of states like this one before Δt_{Saving} elapses and the light turns off.

Fortunately, the difference between the constant state of Formalism B and the continuously changing state of a real-world system can be captured by a single variable: the time Δt_e elapsed since the previous event. If we know that Δt_e has elapsed since motion was last detected, we know that the lighting system will turn off after a time of $\Delta t_{Saving} - \Delta t_e$.

Having acknowledged the importance of the elapsed time, we now have a means to represent two types of state. First we have our original type of state, s, which remains constant between events. We will continue to refer to s as "the state", despite the fact that we now have another type of state. The other type is the ***total state***, $(s, \Delta t_e)$, which reflects the continuously changing state of a real-world system. Note that the total state is simply the state (i.e. the first type of state) and the elapsed time, grouped together.

In the DEVS formalism, a model's output values and state transitions can be considered functions of its total state. As mentioned earlier, this approach to representing state sets DEVS apart from other modeling formalisms. It gives DEVS the generality to represent practically any real-world system that varies in time.

DEVS Atomic Models

It can be convenient to distinguish between atomic models, which are indivisible DEVS models, and coupled models, which are DEVS models composed of other DEVS models. The conventions below are typically associated with atomic models. We will later see that, indirectly, they apply to coupled models as well.

$\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$ is the structure of a DEVS atomic model

X is the set of input values

Y is the set of output values

S is the set of states

$\delta_{ext}: Q \times X \to S$ is the external transition function

$Q = \left\{ (s, \Delta t_e) \middle| \begin{matrix} s \in S \\ 0 \leq \Delta t_e \leq ta(s) \end{matrix} \right\}$ is the set of total states

$\delta_{int}: S \to S$ is the internal transition function

$\lambda: S \to Y \cup \{\emptyset\}$ is the output function

$ta: S \to T$ is the time advance function

$T = \{\Delta t_{int} | 0 \leq \Delta t_{int} \leq \infty\}$ is the set of time durations

The first thing to notice is that instead of one transition function δ, there are two: δ_{ext} and δ_{int}. The ***external transition function*** δ_{ext} is invoked whenever an input is received. Observe that one of its arguments is an input value (some $x \in X$). The ***internal transition function*** δ_{int} is invoked at the same time as the ***output function*** λ, though λ is evaluated before δ_{int} changes the state.

At what simulated time, exactly, are λ and δ_{int} invoked? The answer is provided by the ***time advance function*** ta. Suppose that an event has just occurred. It may have been an external event coinciding with an input and the evaluation of δ_{ext}, or it may have been an internal event coinciding with an output and the evaluation of λ and δ_{int}. Regardless, an internal event will occur after a time of $ta(s)$, provided that no inputs are received beforehand.

We stated earlier that a model's output values and state transitions can be considered functions of its total state. Yet we see above that only δ_{ext} takes the total state $(s, \Delta t_e)$ as an argument. The output function and the internal transition function take as their arguments the state s but not the elapsed time Δt_e. It turns out that passing Δt_e into λ or δ_{int} is unnecessary. Whenever λ and δ_{int} are evaluated, Δt_e must be equal to $ta(s)$. So if the total state is needed during an internal event, one simply evaluates the time advance function and obtains the elapsed time.

At the beginning of this chapter, we illustrated the procedure for discrete time simulation and mentioned that we would later present one for the discrete event approach. A discrete event simulation flowchart based on DEVS is shown in Figure 5.

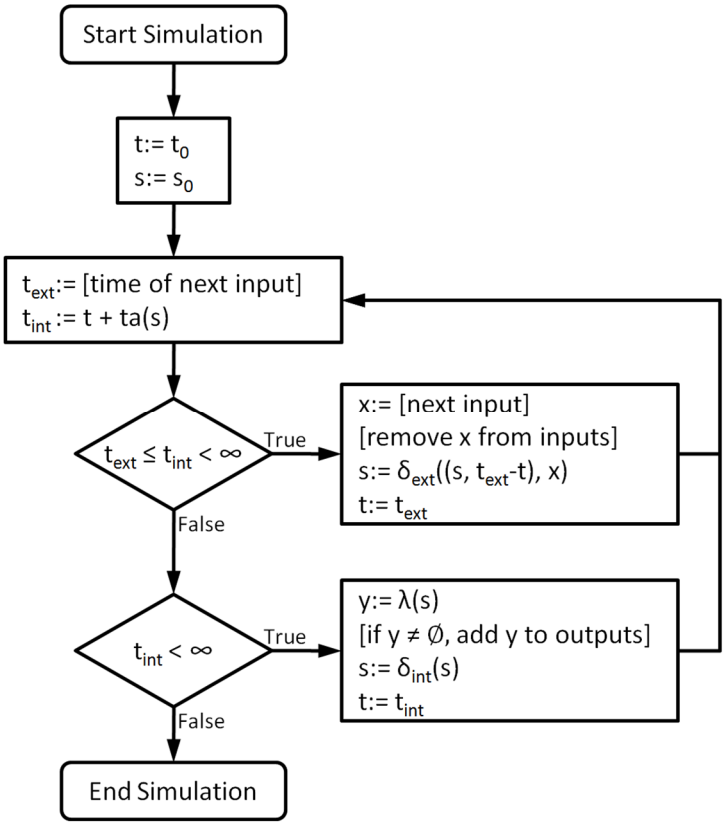

Figure 5. Discrete Event Simulation Procedure using DEVS

Observe that if an input is received before the time elapsed reaches $ta(s)$, the model experiences an external event during which the input is processed. If on the other hand $ta(s)$ elapses before the next input is scheduled, an internal event occurs and an output may be processed. But what happens if the time of the next input coincides with the elapsing of $ta(s)$? Figure 5 indicates that in the case of a tie, external events take priority over internal events. This convention allows one to use the information provided by an input at the earliest possible stage.

For the time being, as we specify DEVS models for the Detector and the Fixture, we will assume that inputs never coincide with the elapsing of $ta(s)$. Later, once coupled models have been introduced, we will see how to ensure that these two models process their outputs before receiving inputs.

Here is a specification of the Detector using the DEVS formalism:

$$Detector = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$

$$X = \{(\text{"action}_{\text{in}}\text{"}, action) | action \in Actions\}$$

$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) | signal \in Signals\}$$

$$S = \left\{ (signal, sent) \middle| \begin{matrix} signal \in Signals \\ sent \in \{\top, \bot\} \end{matrix} \right\}$$

$$\delta_{ext}\Big(\big((signal, sent), \Delta t_e\big), (\text{"action}_{\text{in}}\text{"}, action)\Big) = (signal', sent')$$

$$\begin{pmatrix} action \in \{\text{"Typing", "Waving"}\} \\ signal = \text{"Still"} \end{pmatrix} \Longrightarrow \begin{pmatrix} signal' = \text{"Moving"} \\ sent' = \bot \end{pmatrix}$$

$$\begin{pmatrix} action \in \{\text{"Gone", "Reading"}\} \\ signal = \text{"Moving"} \end{pmatrix} \Longrightarrow \begin{pmatrix} signal' = \text{"Still"} \\ sent' = \bot \end{pmatrix}$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow \begin{pmatrix} signal' = signal \\ sent' = \top \end{pmatrix}$$

$$\delta_{int}\big((signal, \bot)\big) = (signal, \top)$$

$$\lambda\big((signal, \bot)\big) = (\text{"signal}_{\text{out}}\text{"}, signal)$$

$$ta\big((signal, sent)\big) = \Delta t_{int}$$

$$\neg sent \Longrightarrow (\Delta t_{int} = 0)$$

$$sent \Longrightarrow (\Delta t_{int} = \infty)$$

Note that the new δ_{ext} looks a lot like the output function of our Formalism B Detector. One difference is that the resulting signal, either "Moving" or "Still" is recorded in a state variable to be output at a later stage. Another difference is that there is no longer a need to output ∅. Instead, we make use of the new a state variable *sent*, which is either true (⊤) or false (⊥), to avoid unwanted outputs. The following explains how that works.

If an input is received, then after δ_{ext} updates the state, the time advance function will be evaluated. In the case that δ_{ext} changes the signal from "Still" to "Moving" or from "Moving" to "Still", *sent* is assigned ⊥ and consequently $ta(s)$ is 0. The output therefore occurs immediately. Once the output value $\lambda(s)$ is sent, δ_{int} changes *sent* to ⊤. This causes $ta(s)$ to yield ∞, which means nothing happens until the next input arrives.

Suppose that δ_{ext} leaves the signal unchanged (i.e. the "[above conditions are all false]" implication is selected). According to the specification, *sent* must end up ⊤, and thus $ta(s)$ will yield ∞. In this case there is no need for λ to yield ∅, as $ta(s) = \infty$ prevents λ from being evaluated at all.

With the Detector model out of the way, we turn our attention to the Fixture model. Recall that the Fixture emits light in response to a "Moving" signal, and turns the light off in response to a "Still" signal. Also recall that the light only turns off after Δt_{Saving} elapses. Here is the specification:

$$Fixture\big(\Delta t_{Saving}\big) = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$

$$X = \{(\text{"signal}_{\text{in}}\text{"}, signal) | signal \in Signals\}$$

$$Y = \{(\text{"level}_{\text{out}}\text{"}, level) | level \in Levels\}$$

$$S = \left\{ (level, \Delta t_{int}) \middle| \begin{matrix} level \in Levels \\ 0 \leq \Delta t_{int} \leq \infty \end{matrix} \right\}$$

$$\delta_{ext}\Big(\big((level, \Delta t_{int}), \Delta t_e\big), (\text{"signal}_{\text{in}}\text{"}, signal)\Big) = (level, \Delta t_{int}')$$

$$\begin{pmatrix} signal = \text{"Moving"} \\ level = \text{"Dark"} \end{pmatrix} \Longrightarrow (\Delta t_{int}' = 0)$$

$$\begin{pmatrix} signal = \text{"Still"} \\ level = \text{"Light"} \\ \Delta t_{int} = \infty \end{pmatrix} \Longrightarrow \big(\Delta t_{int}' = \Delta t_{Saving}\big)$$

$$\begin{pmatrix} signal = \text{"Still"} \\ level = \text{"Light"} \\ \Delta t_{int} \neq \infty \end{pmatrix} \Longrightarrow (\Delta t_{int}' = \Delta t_{int} - \Delta t_e)$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow (\Delta t_{int}' = \infty)$$

$$\delta_{int}\big((level, \Delta t_{int})\big) = (level', \infty)$$

$$(level = \text{"Dark"}) \Longrightarrow (level' = \text{"Light"})$$

$$(level = \text{"Light"}) \Longrightarrow (level' = \text{"Dark"})$$

$$\lambda\big((level, \Delta t_{int})\big) = (\text{"level}_{\text{out}}\text{"}, level')$$

$$(level = \text{"Dark"}) \Longrightarrow (level' = \text{"Light"})$$

$$(level = \text{"Light"}) \Longrightarrow (level' = \text{"Dark"})$$

$$ta\big((level, \Delta t_{int})\big) = \Delta t_{int}$$

There are two important observations here. First note that one of the state variables, Δt_{int}, directly provides the result of the time advance function. This is a very common technique in the specification of DEVS models. The other observation is the use of Δt_e in δ_{ext}.

When the light turns on, δ_{int} indicates that the Δt_{int} state variable is assigned ∞. If the Fixture model then receives a "Still" signal, it must turn the lights off after a time of Δt_{Saving}. Hence there is a case in δ_{ext} that, upon finding $\Delta t_{int} = \infty$, sets Δt_{int} to Δt_{Saving}. Continuing this scenario, suppose that the Fixture model receives another "Still" signal before Δt_{Saving} elapses. (It is true that we previously went to great trouble to prevent the Detector model from outputting two "Still" signals in a row. However, if we want the Fixture model to be reusable in other contexts, its specification should accommodate any sequence of inputs.) Because this is the second consecutive "Still" signal, it is handled by the case in δ_{ext} that requires $\Delta t_{int} \neq \infty$. In this case the Fixture was previously about to turn off after a time of Δt_{int}, but Δt_e has elapsed since then, so now it must turn off after $\Delta t_{int} - \Delta t_e$. This demonstrates the importance of the elapsed time as an argument of the external transition function.

In this section and the previous, we have looked at three formalisms: Formalism A, Formalism B, and DEVS. Each of these formalisms was more complex than the previous, but allowed us to define a larger set of possible models. Extrapolating this trend, one wonders if there are models that cannot be specified with DEVS. When might we require yet another, even more flexible formalism? The answer is hardly ever. True to its name, DEVS is a very general formalism for specifying discrete event simulation models. Incidentally, it can also be used for discrete time simulation, which is really just a special case of the discrete event approach.

To be fair, while DEVS is a plausible option for modeling almost any time-varying system, it may not be the most convenient option for all applications. If the scope of a simulation project is both constrained and well understood, other approaches should be considered as well. But especially for large projects, it is reassuring to use a set of conventions like DEVS that can accommodate a wide range of potentially unforeseen model requirements.

Research has shown that for any of a great number of alternative modeling formalisms, any specification written in that formalism can be mapped into a DEVS specification. This generality has led to the description of DEVS as a "common denominator" that supports the use of multiple formalisms in a single project (Vangheluwe, 2000).

Legitimacy of Atomic Models

Whenever we specify a model using DEVS, we ought to ensure the specification is both consistent and legitimate. For a specification to be ***consistent***, it must contradict neither itself nor the conventions of the formalism. Suppose we have a DEVS model in which Y is the set of positive real numbers. If there exists an $s \in S$ for which $\lambda(s)$ is negative, the specification is inconsistent.

Although ensuring consistency may require considerable effort, the concept is intuitive and its importance is obvious. Legitimacy is more subtle. Even if a DEVS model has a consistent specification, it will not necessarily allow simulated time to properly advance. The problem is not that the simulation procedure stops. Rather, if the specification is not legitimate, an infinite number of events may occur in a finite duration of simulated time.

A DEVS model has a ***legitimate*** specification if, in the absence of inputs, simulated time will necessarily advance towards ∞ without stopping or converging. This condition can be written mathematically as follows:

$$\sum_{i=0}^{\infty} ta(s_i) = \infty \quad \text{for all} \quad s_0 \in S \quad \text{where for } i \geq 1, \quad s_i = \delta_{int}(s_{i-1})$$

The convenient thing about this condition is that, when assessing the legitimacy of an atomic model, one can ignore a large part of the specification. In fact, only S, δ_{int}, and ta are relevant. To demonstrate, let us check whether our Detector model is legitimate. The relevant part of the specification is repeated below.

$$S = \left\{ (signal, sent) \middle| \begin{matrix} signal \in Signals \\ sent \in \{\top, \bot\} \end{matrix} \right\}$$

$$\delta_{int}\big((signal, \bot)\big) = (signal, \top)$$

$$ta\big((signal, sent)\big) = \Delta t_{int}$$

$$\neg sent \Longrightarrow (\Delta t_{int} = 0)$$

$$sent \Longrightarrow (\Delta t_{int} = \infty)$$

We can see that δ_{int} changes the state variable *sent* to ⊤. If *sent* is ⊤, *ta* yields ∞, so the model is legitimate.

$$\begin{aligned}\sum_{i=0}^{\infty} ta(s_i) &= ta(s_0) + ta\big(\delta_{int}(s_0)\big) + ta\Big(\delta_{int}\big(\delta_{int}(s_0)\big)\Big) + \cdots \\ &= ta(s_0) + ta\big((signal_1, \top)\big) + ta\big((signal_2, \top)\big) + \cdots \\ &= ta(s_0) + \infty + \infty + \cdots \\ &= \infty\end{aligned}$$

Regardless of what state it is in, the Detector model will experience at most one internal event before it starts waiting for an input. We will leave it as an exercise for the reader to show that the Fixture model behaves similarly, and is therefore legitimate as well. Things are more complicated with the Worker model, which can enter into a never-ending cycle of states despite the absence of inputs. The cycle itself is not a problem, but we must ensure that time advances at each repetition.

We have yet to give the complete specification of the Worker model. Here it is:

$$Worker\big(\Delta t_{Typing}, \Delta t_{Reading}\big) = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\rangle$$

$$X = X_{phase} \cup X_{level}$$

$$X_{phase} = \{(\text{"phase}_{\text{in}}\text{"}, phase\ ) \mid phase \in Phases\}$$

$$X_{level} = \{(\text{"level}_{\text{in}}\text{"}, level\ ) \mid level \in Levels\}$$

$$Y = \{(\text{"action}_{\text{out}}\text{"}, action\ ) \mid \text{action} \in \text{action}\}$$

$$S = \left\{ (action, \Delta t_r, sent) \;\middle|\; \begin{array}{c} action \in Actions \\ 0 \le \Delta t_r \le \infty \\ sent \in \{\top, \bot\} \\ (action = \text{"Gone"}) \Longrightarrow (\Delta t_r = \infty) \end{array} \right\}$$

$$\delta_{ext}\Big(\big((action, \Delta t_r, sent), \Delta t_e\big), x\Big) = (action', \Delta t_r', sent')$$

$$\begin{pmatrix} x = (\text{"phase}_{\text{in}}\text{","Present"}) \\ action = \text{"Gone"} \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Typing"} \\ \Delta t_r' = \Delta t_{Typing} \\ sent' = \bot \end{pmatrix}$$

$$\begin{pmatrix} x = (\text{"phase}_{\text{in}}\text{","Absent"}) \\ action \neq \text{"Gone"} \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Gone"} \\ \Delta t_r' = \infty \\ sent' = \bot \end{pmatrix}$$

$$\begin{pmatrix} x = (\text{"level}_{\text{in}}\text{","Dark"}) \\ action = \text{"Reading"} \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Waving"} \\ \Delta t_r' = \Delta t_r - \Delta t_e \\ sent' = \bot \end{pmatrix}$$

$$\begin{pmatrix} x = (\text{"level}_{\text{in}}\text{","Light"}) \\ action = \text{"Waving"} \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Reading"} \\ \Delta t_r' = \Delta t_r - \Delta t_e \\ sent' = \bot \end{pmatrix}$$

$$\begin{bmatrix}\text{above conditions}\\ \text{are all false}\end{bmatrix} \Longrightarrow \begin{pmatrix} action' = action \\ \Delta t_r{}' = \Delta t_r - \Delta t_e \\ sent' = sent \end{pmatrix}$$

$$\delta_{int}\big((action, \Delta t_r, sent)\big) = (action', \Delta t_r{}', \top)$$

$$\neg sent \Longrightarrow \begin{pmatrix} action' = action \\ \Delta t_r{}' = \Delta t_r \end{pmatrix}$$

$$\begin{pmatrix} action = \text{"Typing"} \\ sent \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Reading"} \\ \Delta t_r{}' = \Delta t_{Reading} \end{pmatrix}$$

$$\begin{pmatrix} action \in \{\text{"Reading", "Waving"}\} \\ sent \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Typing"} \\ \Delta t_r{}' = \Delta t_{Typing} \end{pmatrix}$$

$$\lambda\big((action, \Delta t_r, sent)\big) = (\text{"action}_{\text{out}}\text{"}, action')$$

$$\neg sent \Longrightarrow (action' = action)$$

$$\begin{pmatrix} action = \text{"Typing"} \\ sent \end{pmatrix} \Longrightarrow (action' = \text{"Reading"})$$

$$\begin{pmatrix} action \in \{\text{"Reading", "Waving"}\} \\ sent \end{pmatrix} \Longrightarrow (action' = \text{"Typing"})$$

$$ta\big((action, \Delta t_r, sent)\big) = \Delta t_{int}$$

$$\neg sent \Longrightarrow (\Delta t_{int} = 0)$$

$$sent \Longrightarrow (\Delta t_{int} = \Delta t_r)$$

Recall that when the office worker is at their workstation (i.e. after an input phase of “Present” is received), they alternate between “Typing” and “Reading”. When they are reading it is possible for the lights to turn off (i.e. an input level of “Dark” may be received). This causes the Worker to start “Waving” until the light returns. The transition functions above provide a formal description of this behavior.

We encourage the reader to further study how this specification fulfills the informal requirements presented near the beginning of the chapter. But our task at the moment is to determine mathematically whether the specification is legitimate. It will simplify things greatly if we notice that δ_{int} always assigns the state variable *sent* the value $\top$.

$$\delta_{int}\big((action, \Delta t_r, sent)\big) = (action', \Delta t_r{}', \top)$$

With *sent* being $\top$, time is always advanced by the value of the state variable Δt_r.

$$ta\big((action, \Delta t_r, \top)\big) = \Delta t_r$$

According to *S*, it is a possibility that Δt_r is 0. This concerns us because it leaves open the possibility that the key term affecting legitimacy, $ta(\delta_{int}(s))$, is also 0.

$$S = \left\{ (action, \Delta t_r, sent) \;\middle|\; \begin{array}{c} action \in Actions \\ 0 \leq \Delta t_r \leq \infty \\ sent \in \{\top, \bot\} \\ (action = \text{"Gone"}) \Longrightarrow (\Delta t_r = \infty) \end{array} \right\}$$

Note that S also tells us that if $action$ = "Gone", $\Delta t_r = \infty$. If Δt_r is guaranteed to be ∞ at any point, the specification is legitimate. We can therefore focus on the remaining three actions, "Typing", "Reading", and "Waving". These actions are addressed by the following conditions in δ_{int}.

$$\begin{pmatrix} action = \text{"Typing"} \\ sent \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Reading"} \\ \Delta t_r{}' = \Delta t_{Reading} \end{pmatrix}$$

$$\begin{pmatrix} action \in \{\text{"Reading"}, \text{"Waving"}\} \\ sent \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Typing"} \\ \Delta t_r{}' = \Delta t_{Typing} \end{pmatrix}$$

Here we see that Δt_r must be one of two values, Δt_{Typing} or $\Delta t_{Reading}$. And because the state variable $action$ will alternate between "Typing" and "Reading" after at most one event, Δt_r will alternate between Δt_{Typing} and $\Delta t_{Reading}$. With this observation, we can derive an expression for the total accumulated time.

$$\sum_{i=0}^{\infty} ta(s_i) = ta(s_0) + ta\big(\delta_{int}(s_0)\big) + ta\Big(\delta_{int}\big(\delta_{int}(s_0)\big)\Big) + \cdots$$

$$= \begin{pmatrix} ta(s_0) + \Delta t_{Typing} + \Delta t_{Reading} + \Delta t_{Typing} + \Delta t_{Reading} + \cdots \\ \text{or} \\ ta(s_0) + \Delta t_{Reading} + \Delta t_{Typing} + \Delta t_{Reading} + \Delta t_{Typing} + \cdots \end{pmatrix}$$

$$= ta(s_0) + \lim_{n \to \infty} n \cdot \big(\Delta t_{Reading} + \Delta t_{Typing}\big)$$

We do not know the value of $ta(s_0)$. However, we can assure ourselves that it is not necessarily ∞, so we should focus on the second term. To guarantee legitimacy, the second term must be ∞. We can see that this will be the case as long as $(\Delta t_{Typing} + \Delta t_{Reading}) > 0$.

As an aside, note that regardless of legitimacy we should know that $\Delta t_{Typing} \geq 0$ and $\Delta t_{Reading} \geq 0$. Why? For one thing, it makes no sense for the Worker to type or read for a negative duration. But mathematically, a negative Δt_{Typing} or $\Delta t_{Reading}$ would render the Worker model inconsistent. To see this inconsistency, review the DEVS formalism's conventions for the time advance function.

Having assessed the legitimacy of the Worker model, we now realize that a nonnegative Δt_{Typing} and $\Delta t_{Reading}$ is not enough. We now know that their sum must be positive, which means that one of the parameters must be positive. Combined, we have the following constraints.

$$\Delta t_{Typing} \geq 0$$
$$\Delta t_{Reading} \geq 0$$
$$\big(\Delta t_{Reading} > 0\big) \vee \big(\Delta t_{Typing} > 0\big)$$

Whereas the Detector and Fixture specifications were unconditionally legitimate, the Worker specification is legitimate only if we restrict the model parameters. This result makes sense. If Δt_{Typing} and $\Delta t_{Reading}$ are both 0, the Worker will alternative between typing and reading an infinite number of times at a single instant of simulated time. Assessing legitimacy is useful, as dangers like $\Delta t_{Typing} = \Delta t_{Reading} = 0$ can go undiscovered until after simulation software is deployed. Another useful discovery from the above analysis is the fact that if exactly one of the two parameters is 0, the specification is legitimate. Our model can represent an office worker who only spends time typing, or only spends time reading.

If a time advance function always yields a positive value, may we assume that the specification is legitimate? The intuitive answer is yes, but the correct answer is no. Consider the following partial specification of the model Pitfall:

$$Pitfall = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$
$$S = \{\Delta t \mid 0 < \Delta t \leq \infty\}$$
$$\delta_{int}(\Delta t) = \frac{\Delta t}{2}$$
$$ta(\Delta t) = \Delta t$$

Here the result of *ta* is the value of the one and only state variable Δt. According to *S*, Δt may never be 0, so we know that time will advance between every internal event. The guaranteed advancement of time suggests legitimacy, unless it turns out that the sum of the time intervals converges. Here δ_{int} scales down Δt by a factor of 2 at each event. Decreasing at this rate, the intervals will in fact converge. If the initial value of the state variable is Δt_0, and if there are no intervening inputs, the simulation will progress no further than twice Δt_0.

$$\sum_{i=0}^{\infty} ta(s_i) = ta(\Delta t_0) + ta\big(\delta_{int}(\Delta t_0)\big) + ta\left(\delta_{int}\big(\delta_{int}(\Delta t_0)\big)\right) + \cdots$$
$$= \Delta t_0 + \delta_{int}(\Delta t_0) + \delta_{int}\big(\delta_{int}(\Delta t_0)\big) + \cdots$$
$$= \Delta t_0 + \frac{\Delta t_0}{2} + \delta_{int}\left(\frac{\Delta t_0}{2}\right) + \cdots$$
$$= \Delta t_0 + \frac{\Delta t_0}{2} + \frac{\Delta t_0}{4} + \cdots$$
$$= \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \cdot \Delta t_0$$
$$= 2 \cdot \Delta t_0$$

The set of states permits $\Delta t_0 < \infty$, in which case $2 \cdot \Delta t_0 < \infty$; therefore, the Pitfall specification is not legitimate.

Because the Worker model has only discrete state variables and none of them are unbounded, the model has a finite number of possible states. For such a model, legitimacy requires that every

possible cycle of states contains at least one state s for which $ta(s) > 0$. The relevant cycle in the Worker model was the "Typing"/"Reading" cycle, and indeed we found that a positive duration is required for at least one of these two states.

Note that this "cycle of states" rule is insufficient for models like Pitfall that have an infinite number of possible states. For these types of models, even if $ta(s) > 0$ for all s, the specification may still not be legitimate. In such cases, one technique for ensuring legitimacy is to require $ta(\delta_{int}(s)) > \Delta t_\varepsilon$ for all $s \in S$, where Δt_ε is some small but positive constant duration.

DEVS Coupled Models

At this point we have an atomic model specification for the Detector, the Fixture, and the Worker. To complete a specification of the Automatic Lighting Environment, we need only link the atomic models together as submodels of a DEVS coupled model. The conventions for doing this are below.

$\langle X, Y, D, M, EIC, EOC, IC, Select \rangle$ is the structure of a DEVS coupled model

X is the set of input values

Y is the set of output values

D is the set of submodel IDs

$M: D \to \mathcal{M}$ is the ID-to-submodel mapping function

$\mathcal{M}$ is the set of possible DEVS models

EIC is the set of external input couplings

EOC is the set of external output couplings

IC is the set of internal couplings

$Select: 2^D \to D$ is the tie-breaking function

We can see that, like an atomic model, a coupled model has a set of inputs X and a set of outputs Y. Coupled models also have ports associated with their inputs and outputs. We will give our Environment model an input port for the two possible phases in the office worker's schedule, "Present" and "Absent". We will also define two output ports, one for the office worker's actions and one for the lighting level. These outputs contain information relevant to the performance of the automatic lighting system. An occurrence of the "Waving" action indicates inconvenience for an office worker left momentarily in the dark, and any time elapsed while the lights are on and the office worker is "Gone" could be viewed as a waste of electricity.

The input and output ports of the Environment model are shown in Figure 6, along with the relationships between these ports and the ports of the three submodels.

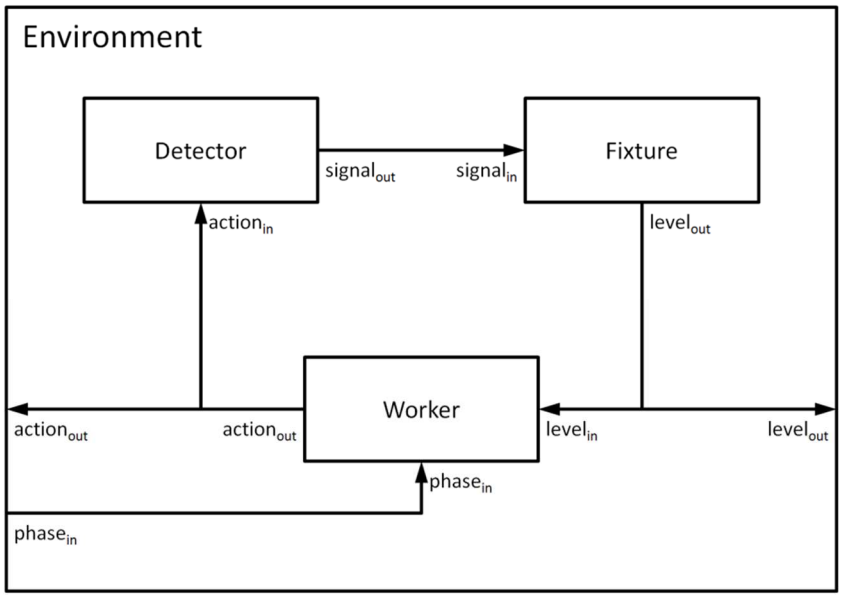

Figure 6. Coupled Model of an Automatic Lighting Environment

Below is the specification of the Environment model, with the exception of the tie-breaking function. We will look at *Select* later.

$$Environment\left(\Delta t_{Saving}, \Delta t_{Typing}, \Delta t_{Reading}\right) = \langle X, Y, D, M, EIC, EOC, IC, Select\rangle$$

$$X = \{(\text{"phase}_{\text{in}}\text{"}, phase) | phase \in Phases\}$$

$$Y = Y_{action} \cup Y_{level}$$

$$Y_{action} = \{(\text{"action}_{\text{out}}\text{"}, action) | action \in Actions\}$$

$$Y_{level} = \{(\text{"level}_{\text{out}}\text{"}, level) | level \in Levels\}$$

$$D = \{\text{"Detector"}, \text{"Fixture"}, \text{"Worker"}\}$$

$$M(d) = m$$

$$(d = \text{"Detector"}) \Longrightarrow (m = Detector)$$

$$(d = \text{"Fixture"}) \Longrightarrow \left(m = Fixture\left(\Delta t_{Saving}\right)\right)$$

$$(d = \text{"Worker"}) \Longrightarrow \left(m = Worker\left(\Delta t_{Typing}, \Delta t_{Reading}\right)\right)$$

$$EIC = \{((\text{""}, \text{"phase}_{\text{in}}\text{"}), (\text{"Worker"}, \text{"phase}_{\text{in}}\text{"}))\}$$

$$EOC = \left\{\begin{array}{c}((\text{"Fixture"}, \text{"level}_{\text{out}}\text{"}), (\text{""}, \text{"level}_{\text{out}}\text{"})),\\ ((\text{"Worker"}, \text{"action}_{\text{out}}\text{"}), (\text{""}, \text{"action}_{\text{out}}\text{"}))\end{array}\right\}$$

$$IC = \left\{\begin{array}{c}((\text{"Worker"}, \text{"action}_{\text{out}}\text{"}), (\text{"Detector"}, \text{"action}_{\text{in}}\text{"})),\\ ((\text{"Detector"}, \text{"signal}_{\text{out}}\text{"}), (\text{"Fixture"}, \text{"signal}_{\text{in}}\text{"})),\\ ((\text{"Fixture"}, \text{"level}_{\text{out}}\text{"}), (\text{"Worker"}, \text{"level}_{\text{in}}\text{"}))\end{array}\right\}$$

Note that a large part of this specification is merely a formal representation of the information in Figure 6. The set *D* contains a unique ID for each submodel, and here we have used the same labels as in the diagram. The empty string "" serves as the ID of the coupled model itself (sometimes "Self" or other symbols are used).

Each coupling between ports takes the form (([source ID], [output port]), ([destination ID], [input port])). The diagram shows that the Environment's one input port, "phase_{in}", is connected to the "phase_{in}" input port of the "Worker" submodel. The port names match in this case, but they need not. The relationship is represented by (("","phase_{in}"), ("Worker", "phase_{in}")) in the set *EIC*. Similarly, the links in the diagram between the submodels and the Environment's output ports can be found in *EOC*, and the links from submodel to submodel are in *IC*.

The variable *M* specifies the DEVS model associated with each submodel ID. Observe that we have used its definition to distribute the parameters of the Environment model to the individual submodels. Here we are treating *M* as a function that maps an ID *d* to the corresponding DEVS submodel $M(d) = \langle X_d, Y_d, \ldots \rangle$. Note that *M* is often defined instead as a set of submodels M_d. Either way, consistency requires that for every coupling, the possible values for the source's output port constitute a subset of the possible values for the destination's input port.

Earlier we presented the simulation procedure associated with DEVS atomic models. Now that we are missing the transition functions, the output function, and the time advance function, what is the simulation procedure for coupled models? It turns out that the procedure is the same. DEVS has a property known as ***closure under coupling***, which guarantees that the behavior of any coupled model can be represented using the conventions associated with atomic models. Here again are both sets of conventions:

$\langle X, Y, D, M, EIC, EOC, IC, Select \rangle$ is the structure of a DEVS coupled model

$\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$ is the structure of a DEVS atomic model

The 8 variables that compose a coupled model can be mapped into the 7 variables of an atomic model, yielding what is referred to as the ***resultant***. We will review this mapping informally to highlight the sequences of events that occur in coupled models. First, the input and output sets *X* and *Y* are the same in both a coupled model and its resultant. The state of the resultant includes the total state of every model in *M*. Therefore, the set of states *S* includes all possible combinations of all possible total states for every submodel.

A coupled model experiences an external event when it receives an input. In that case, the resultant's δ_{ext} redirects the input to all receiving submodels as specified by *EIC*. Each receiving submodel then experiences its own external event; their δ_{ext} functions are invoked. For example, a "phase_{in}" input received by the Environment model will get redirected to the "phase_{in}" port of the Worker model. The Worker model will then receive the same input and experience an external transition.

The resultant's *ta* yields the time before any one submodel experiences an internal event. If this time elapses, the coupled model experiences an internal event as well. The resultant's λ and δ_{int}

invoke the λ and δ_{int} functions associated with the one submodel that triggered the event. The triggering submodel's output is redirected to receiving submodels according to *IC*, and those receiving submodels experience external events. For example, if the Detector model triggers an internal event, a "$\text{signal}_{\text{out}}$" output will be sent to the Fixture model. The Fixture model will then experience an external event.

If, according to *EOC*, the triggering submodel's output is linked to the output of the entire coupled model, then the resultant's λ reflects that output. Otherwise, the resultant's λ yields ∅. So if the Fixture model sends an output, the Environment model sends the same output as well. But if the Detector model sends an output, the Environment model outputs ∅.

Note that when an event of any kind occurs in a coupled model, the elapsed time associated with every submodel is updated.

That mostly describes the behavior of a coupled model, though there is one remaining complication: multiple submodels may try to trigger internal events at the same time. In such cases, the ***select function*** is used to break the tie. The function takes the argument D_{imm}, the set of IDs of all imminent submodels. A submodel is ***imminent*** if it is scheduled to experience an internal event at least as soon as any other. The result of *Select* is d_s, the ID of the submodel selected to trigger the internal event ($d_s \in D_{imm}$). Here is the select function for the Environment model:

$$
\begin{aligned}
&Select(D_{imm}) = d_s \\
&\quad (\text{"Fixture"} \in D_{imm}) \Longrightarrow (d_s = \text{"Fixture"}) \\
&\quad \begin{pmatrix} \text{"Detector"} \in D_{imm} \\ \text{"Fixture"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"Detector"}) \\
&\quad \begin{pmatrix} \text{"Worker"} \in D_{imm} \\ \text{"Detector"} \notin D_{imm} \\ \text{"Fixture"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"Worker"})
\end{aligned}
$$

When we first specified DEVS atomic models for the Detector and the Fixture, we assumed that neither model would ever have an input coincide with the elapsing of $ta(s)$. In other words, we would never have to choose between an external event and an internal event for these models. This select function validates that assumption, at least in the context of the Environment model. For example, suppose the Detector and Fixture submodels are both imminent. According to *Select*, the Fixture model experiences the internal event first. By the time the Detector model triggers an internal event and signals the Fixture model, the Fixture model is in a new state and is no longer imminent. In a similar fashion, *Select* prevents the Worker model from sending actions to an imminent Detector model.

One cannot always rely on *Select* to prevent collisions between external and internal events. The Worker model in the "Reading" state may well receive a "Dark" input at the same time that it is scheduled to transition to "Typing". Will the Worker start "Waving" in response to the loss of light, or will it simply start "Typing"? According to the simulation procedure we presented for DEVS models, external events take priority. So the Worker will enter the "Waving" state immediately, revert to "Reading" in response to another input when the lights come back on, and only then enter

the “Typing” state. We will assume this behavior is acceptable, but otherwise we would modify the Worker model specification.

It is common practice to define the select function with a list of submodel IDs; for example, (“Fixture”, “Detector”, “Worker”). Whenever there is a tie, the submodel closest to the front of the list is selected. A list offers less flexibility than a function. But in many cases, including our Environment model, it would suffice.

Legitimacy of Coupled Models

When we say a coupled model is legitimate, we mean that its resultant is legitimate based on the definition presented earlier for atomic models. As one would expect, for a coupled model to be legitimate, all of its submodels must be legitimate. The question is, if all of its submodels are legitimate, may we assume that the coupled model is legitimate? It turns out that if there are no feedback loops in the coupled model, the answer is yes. But if there are feedback loops, we have more work to do.

A ***feedback loop*** in a coupled model is any circular path formed by traversing couplings from their source submodels to their destination submodels. There is one feedback loop in the Environment model, as the Worker sends outputs to the Detector which outputs to the Fixture which outputs to the Worker. The problem is not the existence of a feedback loop, but rather the possibility that a sequence of self-perpetuating events propagates around the loop an infinite number of times in a finite duration of time.

Recall that when assessing the legitimacy of an atomic model, one generally studies the value of $ta(\delta_{int}(s))$. Here we are considerably more interested in $ta(\delta_{ext}((s, \Delta t_e), x))$, the delay between receiving an input and sending an output. Why? The time required for a sequence of events to propagate around a feedback loop is the sum of these delays. Suppose the circular propagation of events repeats itself indefinitely. If all of these $ta(\delta_{ext}((s, \Delta t_e), x))$ delay values either equal 0 or their sum converges on 0, the model is not legitimate.

In the case of the Environment model, all submodels have a finite number of states. With this type of model, the only concern is the possibility that all $ta(\delta_{ext}((s, \Delta t_e), x))$ values equal 0. Let us start with the Worker model, and determine all cases in which there is no delay. Because we are worried about cycles of events with no time advancement whatsoever, we may simplify matters by assuming the elapsed time Δt_e is 0. Also, although the Worker model has two input ports, the “phase_{in}” is not part of the feedback loop and can be ignored. With these simplifications, here is the Worker model’s external transition function:

$$\delta_{ext}\Big(((action, \Delta t_r, sent), 0), (\text{"level}_{\text{in}}\text{"}, level)\Big) = (action', \Delta t_r', sent')$$

$$\begin{pmatrix} level = \text{"Dark"} \\ action = \text{"Reading"} \end{pmatrix} \Rightarrow \begin{pmatrix} action' = \text{"Waving"} \\ \Delta t_r' = \Delta t_r \\ sent' = \perp \end{pmatrix}$$

$$\begin{pmatrix} level = \text{"Light"} \\ action = \text{"Waving"} \end{pmatrix} \Longrightarrow \begin{pmatrix} action' = \text{"Reading"} \\ \Delta t_r{}' = \Delta t_r \\ sent' = \bot \end{pmatrix}$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow \begin{pmatrix} action' = action \\ \Delta t_r{}' = \Delta t_r \\ sent' = sent \end{pmatrix}$$

We must also look at the Worker model's time advance function.

$$ta\big((action, \Delta t_r, sent)\big) = \Delta t_{int}$$

$$\neg sent \Longrightarrow (\Delta t_{int} = 0)$$

$$sent \Longrightarrow (\Delta t_{int} = \Delta t_r)$$

Substituting the result of the external transition function into the time advance function, we get a formula for the delay Δt_{int} between the Worker model's inputs and outputs.

$$ta\left(\delta_{ext}\left(\big((action, \Delta t_r, sent), 0\big), (\text{"level}_{\text{in}}\text{"}, level)\right)\right) = \Delta t_{int}$$

$$\begin{pmatrix} level = \text{"Dark"} \\ action = \text{"Reading"} \end{pmatrix} \Longrightarrow (\Delta t_{int} = 0)$$

$$\begin{pmatrix} level = \text{"Light"} \\ action = \text{"Waving"} \end{pmatrix} \Longrightarrow (\Delta t_{int} = 0)$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow \Big(\Delta t_{int} = ta\big((action, \Delta t_r, sent)\big)\Big)$$

There are 3 conditions in total, but the third one is uninteresting since the state of the model has been left unchanged. So effectively there are 2 cases to consider in which $\Delta t_{int} = 0$.

If we perform the same exercise for the Detector model, the delay is given by the following:

$$ta\left(\delta_{ext}\left(\big((signal, sent), 0\big), (\text{"action}_{\text{in}}\text{"}, action)\right)\right) = \Delta t_{int}$$

$$\begin{pmatrix} action \in \{\text{"Typing"}, \text{"Waving"}\} \\ s = \text{"Still"} \end{pmatrix} \Longrightarrow (\Delta t_{int} = 0)$$

$$\begin{pmatrix} action \in \{\text{"Gone"}, \text{"Reading"}\} \\ s = \text{"Moving"} \end{pmatrix} \Longrightarrow (\Delta t_{int} = 0)$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow (\Delta t_{int} = \infty)$$

Again there are 3 conditions, but we may ignore the third one because the resulting delay can never be 0. So effectively we have another 2 cases.

Below is the delay for the Fixture model:

$$ta\left(\delta_{ext}\left(\big((level, \Delta t_{int}), 0\big), (\text{"signal}_{\text{in}}\text{"}, signal)\right)\right) = \Delta t_{int}$$

$$\begin{pmatrix} signal = \text{"Moving"} \\ level = \text{"Dark"} \end{pmatrix} \Longrightarrow (\Delta t_{int} = 0)$$

$$\begin{pmatrix} signal = \text{"Still"} \\ level = \text{"Light"} \\ \Delta t_{int} = \infty \end{pmatrix} \Longrightarrow \left(\Delta t_{int} = \Delta t_{Saving}\right)$$

$$\begin{pmatrix} signal = \text{"Still"} \\ level = \text{"Light"} \\ \Delta t_{int} \neq \infty \end{pmatrix} \Longrightarrow \left(\Delta t_{int} = ta\big((level, \Delta t_{int})\big)\right)$$

$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow (\Delta t_{int} = \infty)$$

Now there are 4 conditions, but again only 2 of them are relevant. For the third condition, the state is unchanged. For the fourth, the delay is never 0. Although the delay in the second condition is not necessarily 0, we have yet to rule out $\Delta t_{Saving} = 0$.

Now we must consider all combinations of states that may lead to a perpetual cycle of events with no delay. The task seems daunting, but fortunately we can neglect all combinations of states that fail to satisfy any of the 6 conditions above for which $\Delta t_{int} = 0$ is a possibility. The first condition in the Worker's Δt_{int} formula suggests a potential problem if the Fixture has become "Dark" while the Worker is "Reading". In that case, the Worker will start "Waving". Looking at the Detector model's Δt_{int}, a "Waving" input is only dangerous if the state is "Still". So we consider the initial combination of states in which the Worker's *action* is "Reading", the Detector's *signal* is "Still", and the Fixture's *level* has just become "Dark". Figure 7 shows the sequence of states resulting from these initial conditions.

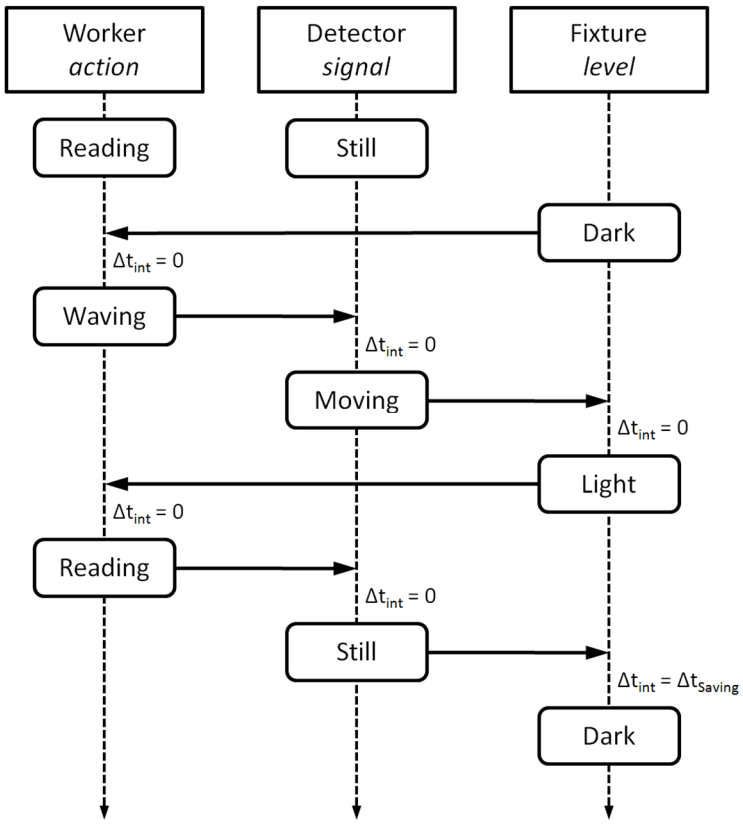

Figure 7. Cycle of States in the Environment Model's Feedback Loop

The "Reading", "Still", "Dark" combination of states does indeed lead to a self-perpetuating cycle of events. The office worker waves to trigger the lighting system, the lights turn on, the office worker resumes reading, the lights eventually turn off, and the cycles repeats. Figure 7 shows the state of each submodel at each stage in the cycle. The key thing to note is that the submodels end up in the original combination of states, which produces the repetition.

It so happens that this cycle involves all 6 of the relevant cases in the delay formulas above. Furthermore, if we consider each case one by one like we did for the first condition in the Worker's Δt_{int} formula, we will end up with the same sequence of states. The initial combination of states may differ, but the cycle will be the same. Therefore, the legitimacy of the Environment model depends only on whether time advances at all during this cycle.

Notice for each step in the cycle, Figure 7 shows the delay between receiving an input and sending the resulting output. The delay values came directly from the 6 cases in the formulas for Δt_{int}. According to these values, it takes a time of Δt_{Saving} for events to propagate twice around the feedback loop, or for one repetition of the cycle of states. Therefore, the Environment model is only legitimate if $\Delta t_{Saving} > 0$.

This result is not particularly intuitive, as there is nothing fundamentally wrong with modeling a lighting fixture that turns off immediately when no motion is detected. In fact, if we modify the Worker model to include a positive delay before transitioning from "Reading" to "Waving", Δt_{Saving} =

0 would yield a legitimate Environment model. This might be a worthwhile exercise for the reader. The important point is that DEVS provides techniques to assess the legitimacy of both atomic and coupled model specifications. Had we not analyzed the specifications in our example, we may not have become aware of the necessary constraints on the Environment model's parameters:

$$\Delta t_{Saving} > 0$$
$$\Delta t_{Typing} \geq 0$$
$$\Delta t_{Reading} \geq 0$$
$$(\Delta t_{Reading} > 0) \vee (\Delta t_{Typing} > 0)$$

Hierarchical Models

Closure under coupling tells us that we could have described the Automatic Lighting Environment as an atomic model instead of a coupled model. However, the modular design approach we adopted allowed us to avoid this potentially difficult task. Instead of attempting to produce one complex atomic model for the Environment system, we specified and combined 3 simpler atomic models.

For extremely complicated systems, even the task of coupling submodels can be problematic. Imagine, for example, the complexity of a coupled model composed of several dozen distinct atomic models. In these situations one should consider a hierarchical approach to model design. Because every coupled model has a resultant, which is essentially an atomic model, one coupled model can be a submodel of another. This nesting of coupled models produces a hierarchy. Note that when replacing the flat structure of a single coupled model with the multi-leveled structure of a hierarchical model, the total number of atomic models remains unchanged. We are simply trading one complex coupled model for several simpler coupled models.

Let us return once again to our Environment model specification. Previously we used a single coupled model composed of 3 atomic models. Here we provide an alternative specification featuring a hierarchical structure. The Environment model will now consist of only 2 submodels, the Worker model and a new Lighting model that represents the Automatic Lighting System. The Lighting model is also specified as a coupled model, and it consists of the Detector model and the Fixture model. The new model structure, with additional ports and slightly different relationships, is shown in Figure 8.

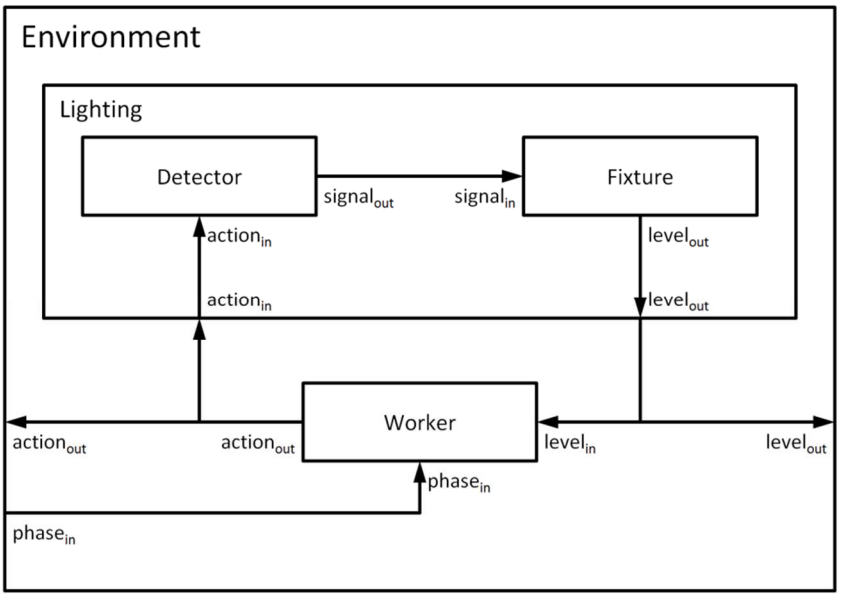

Figure 8. Hierarchical Model of an Automatic Lighting Environment

The conventions for specifying hierarchical models are exactly the same as those for specifying coupled models; we need only apply these conventions multiple times. For our example, we first produce the specification of the Lighting model.

$$Lighting(\Delta t_{Saving}) = \langle X, Y, D, M, EIC, EOC, IC, Select \rangle$$

$$X = \{(\text{"action}_{\text{in}}\text{"}, action) | action \in Actions\}$$

$$Y = \{(\text{"level}_{\text{out}}\text{"}, level) | level \in Levels\}$$

$$D = \{\text{"Detector"}, \text{"Fixture"}\}$$

$$M(d) = m$$

$$(d = \text{"Detector"}) \Longrightarrow (m = Detector)$$

$$(d = \text{"Fixture"}) \Longrightarrow \left(m = Fixture(\Delta t_{Saving})\right)$$

$$EIC = \{((\text{""}, \text{"action}_{\text{in}}\text{"}), (\text{"Detector"}, \text{"action}_{\text{in}}\text{"}))\}$$

$$EOC = \{((\text{"Fixture"}, \text{"level}_{\text{out}}\text{"}), (\text{""}, \text{"level}_{\text{out}}\text{"}))\}$$

$$IC = \{((\text{"Detector"}, \text{"signal}_{\text{out}}\text{"}), (\text{"Fixture"}, \text{"signal}_{\text{in}}\text{"}))\}$$

$$Select(D_{imm}) = d_s$$

$$(\text{"Fixture"} \in D_{imm}) \Longrightarrow (d_s = \text{"Fixture"})$$

$$\begin{pmatrix} \text{"Detector"} \in D_{imm} \\ \text{"Fixture"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"Detector"})$$

Now we provide an alternative specification of the Environment model. Note that the set of input values *X* and the set of output values *Y* are the same as in the previous version, the one coupling all

3 atomic models directly. The fact that the Lighting submodel now replaces the Detector and Fixture submodels does not change the behavior represented by the specification.

$$Environment\left(\Delta t_{Saving}, \Delta t_{Typing}, \Delta t_{Reading}\right) = \langle X, Y, D, M, EIC, EOC, IC, Select\rangle$$

$$X = \{(\text{"phase}_{\text{in}}\text{"}, phase) | phase \in Phases\}$$

$$Y = Y_{action} \cup Y_{level}$$

$$Y_{action} = \{(\text{"action}_{\text{out}}\text{"}, action) | action \in Actions\}$$

$$Y_{level} = \{(\text{"level}_{\text{out}}\text{"}, level) | level \in Levels\}$$

$$D = \{\text{"Lighting"}, \text{"Worker"}\}$$

$$M(d) = m$$

$$(d = \text{"Lighting"}) \Longrightarrow \left(m = Lighting\left(\Delta t_{Saving}\right)\right)$$

$$(d = \text{"Worker"}) \Longrightarrow \left(m = Worker\left(\Delta t_{Typing}, \Delta t_{Reading}\right)\right)$$

$$EIC = \{((\text{""}, \text{"phase}_{\text{in}}\text{"}), (\text{"Worker"}, \text{"phase}_{\text{in}}\text{"}))\}$$

$$EOC = \begin{Bmatrix} ((\text{"Lighting"}, \text{"level}_{\text{out}}\text{"}), (\text{""}, \text{"level}_{\text{out}}\text{"})), \\ ((\text{"Worker"}, \text{"action}_{\text{out}}\text{"}), (\text{""}, \text{"action}_{\text{out}}\text{"})) \end{Bmatrix}$$

$$IC = \begin{Bmatrix} ((\text{"Worker"}, \text{"action}_{\text{out}}\text{"}), (\text{"Lighting"}, \text{"action}_{\text{in}}\text{"})), \\ ((\text{"Lighting"}, \text{"level}_{\text{out}}\text{"}), (\text{"Worker"}, \text{"level}_{\text{in}}\text{"})) \end{Bmatrix}$$

$$Select(D_{imm}) = d_s$$

$$(\text{"Lighting"} \in D_{imm}) \Longrightarrow (d_s = \text{"Lighting"})$$

$$\begin{pmatrix} \text{"Worker"} \in D_{imm} \\ \text{"Lighting"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"Worker"})$$

In summary, a simple system can be effectively specified using a single atomic model. Given a more complex system, one may benefit from a modular approach in which the single atomic model is replaced with a single coupled model containing several atomic models. For an even more complex system, one may couple models in a hierarchical fashion. The single coupled model is then replaced with several coupled models nested within one another.

DEVS and Object-Oriented Programming

Support for modular and hierarchical model design is one of the DEVS formalism's most compelling attributes. It is often remarked, however, that widely adopted object-oriented programming practices provide the same benefits. Technically, object-orientation and DEVS are not alternatives to one another. Many simulations have been implemented using object-oriented programming features in conjunction with DEVS conventions. That said, the analogy between object-oriented classes and DEVS models deserves some discussion.

In an objected-oriented language, classes include methods that take arguments, deliver return values, and reassign member variables. Similarly, DEVS models include transition functions that take inputs, deliver outputs, and reassign state variables. The difference is that these transition functions also depend on the simulated time elapsed since the previous event. It is possible to include this temporal information in object-oriented code, but it is not the norm.

The other major difference between object-orientation and DEVS is that, with the former, it is common practice to design classes that reference one another explicitly. For example, one might implement a Detector class that invokes a `receive_signal` method on a reference to a Fixture object. With DEVS, models almost never reference the other models with which they interact. In our Detector specification, there is no mention of the Fixture it was designed to influence. The Detector model simply outputs a signal with no particular destination, and it is up to the encompassing coupled model to direct the signal to the Fixture model. Some pairs of interacting object-oriented classes exhibit a similar degree of independence. But typically, at least one of these classes will depend on the other.

The pseudocode below illustrates an object-oriented but DEVS-unaware implementation of the Detector model. Note that there is no representation of time in the `receive_action` method, the main function responsible for state transitions. Also note that the lighting fixture is explicitly referenced.

```
class Detector_OO:

    private variable signal

    private variable fixture

    public method set_fixture(f):
        fixture:= f

    public method receive_action(action):
        new_signal:= [...]
        if not signal = new_signal:
            signal:= new_signal
            fixture.receive_signal(signal)
```

Compare the implementation above to the following Detector implementation which combines DEVS with object-oriented programming. Time is now represented. In the `external_transition` function, the time elapsed since the previous event is given by the argument `elapsed`. If the elapsed time were needed in the `internal_transition` function, it could be obtained by invoking `time_advance`. Note that there is no longer an explicit reference to the lighting fixture. It is up to the coupled model, which would be implemented in a separate class, to ensure that the result of the `output` method is delivered to a lighting fixture object.

```
class Detector_DEVS inherits from AtomicModel:

    private variable signal

    private variable sent

    public method external_transition(elapsed, action):
        signal, sent:= [...]

    public method internal_transition():
        sent:= true

    public method output():
        return ["signal_out", signal]

    public method time_advance():
        if not sent:
            dt:= 0
        else:
            dt:= infinity
        return dt
```

Our DEVS-aware Detector class inherits from a base class named `AtomicModel`. The idea is that a generic simulation procedure can be implemented once for `AtomicModel`, and applied to any application-specific derived class like `Detector_DEVS`. Admittedly, this use of inheritance does not require a well-established modeling formalism. However, by following DEVS conventions, an object-oriented programmer ensures that his or her simulation code is sufficiently generic to accommodate any discrete-event simulation.

DEVS and Parallel Computing

In what ways can one exploit multi-core processors, multi-process computers, and multi-computer networks to accelerate computer simulations developed with DEVS? One option is to parallelize only the few most time-consuming functions. The drawback to this approach is that, given a large coupled model, it requires the execution time to be dominated by only a few of the many atomic models. Furthermore, the parallelization effort is invested in certain models while others will show no improvement. Another option is to run multiple simulations simultaneously with different sets of parameters, as is frequently done for Monte Carlo or cost minimization problems. Unfortunately, this technique is of little use if one must accelerate a single simulation run. A third option is to exploit the modularity of DEVS to automate the parallelization of a coupled model. For this approach, DEVS practitioners often adopt a variant of the formalism called Parallel DEVS (Chow & Zeigler, 1994).

Recall that the select function orders the internal events of imminent submodels. In our example, if the Detector and the Fixture were scheduled to experience internal events at the same time, we ensured that the Fixture's transition would occur first. Parallel DEVS eliminates the select function. The order of simultaneous events is deliberately left ambiguous to allow the functions of multiple imminent submodels to be executed concurrently.

With Parallel DEVS, all imminent models send outputs simultaneously. This raises two issues which the variant addresses. First note that the outputs of multiple imminent submodels may be directed to a single receiving model. This receiving model must recognize that the order in which its inputs arrive is essentially arbitrary and best ignored. For that reason, the external transition function of a Parallel DEVS atomic model takes a bag of inputs instead of a single input. A bag is like a set in that its items are unordered, but different in that it can contain duplicate items.

The other issue raised by simultaneous outputs is the possibility that one submodel sends an output to a second submodel at the same time that the second submodel sends its own output. Clearly the output functions get evaluated for both models. But since the second model is both receiving an input and sending an output, which transition function should be called? In many cases it makes sense to call δ_{int} first, since the output function has already been evaluated, and then to call δ_{ext} to process the input. But that raises the question of what elapsed time value to pass into δ_{ext}: should it be 0 or the previous $ta(s)$? Parallel DEVS addresses the issue by requiring atomic models to have a ***confluent transition function*** δ_{con}. It is invoked in place of δ_{ext} or δ_{int} whenever external and internal events collide.

Earlier in the chapter when we specified atomic models for the Detector and Fixture, we assumed that external and internal events would never collide. We later used *Select* to guarantee that internal events would be fully processed before inputs were received. Had we used Parallel DEVS, we would have defined both confluent transition functions as follows to achieve the same effect.

$$\delta_{con}\left((s, ta(s)), x_{bag}\right) = \delta_{ext}\left((\delta_{int}(s), 0), x_{bag}\right)$$

Summary and Further Reading

With state transitions that depend in part on the time elapsed since the previous event, a DEVS model can represent practically any real-world system that varies in time. The DEVS formalism provides first and foremost a set of conventions for specifying atomic models, along with a procedure for performing simulations with these models. If one specifies a coupled model, then due to closure under coupling one has implicitly defined an equivalent atomic model. This modular approach can be used to avoid a complex atomic model in favor of multiple simpler atomic models. Similarly, by combining models in a hierarchical fashion, one avoids a complex coupled model in favor of multiple simpler coupled models.

It is important to ensure that every DEVS model has a legitimate specification, one that always allows a simulation to properly advance time. For atomic models, this requires an examination of the delay between events in an infinite sequence of internal events. For coupled models, one must look at the delay between inputs and outputs for every submodel in a feedback loop.

We have applied these core DEVS concepts by developing and analyzing specifications representing an office lighting system and its various components. As mentioned at the outset, more information on the DEVS formalism and related theory can be found in Zeigler et al (2000).

It is helpful to understand the similarities and differences between the DEVS formalism and the conventions used by modern software developers. We have already compared DEVS with object-orientation, but another noteworthy set of conventions is the Unified Modeling Language (UML). Traditional UML is somewhat limited in its ability to represent the timing of events. This shortcoming is partially addressed by an extension of UML designed for real-time software systems (UML-RT). Huang & Sarjoughian (2004) provide a detailed comparison of UML-RT with DEVS, and explain how the DEVS formalism's treatment of time is better suited to simulation studies.

DEVS users should familiarize themselves with several variants of the formalism. One of these variants is Parallel DEVS, which we have already discussed. Another notable variant is Cell-DEVS, which applies DEVS to models composed of an array of cells (Wainer & Giambiasi, 2001). Among other things, Cell-DEVS has been used to model the spread of forest fires, the diffusion of heat, and urban traffic. Stochastic DEVS (STDEVS) is one of several ways one can introduce randomness into a DEVS model (Castro et al, 2008). It replaces the deterministic results of the transition functions with probability spaces. There is also a variant called Dynamic Structure DEVS (DSDEVS), which allows a coupled model's submodels and connections to be added and deleted during a simulation (Barros, 1995).

Several noteworthy books cover DEVS from different perspectives. Written for simulation practitioners, Wainer (2009) demonstrates the application of DEVS and the Cell-DEVS variant to physical, biological, environmental, communication, and urban systems. Nutaro (2011) focuses on the implementation of simulation software using object-oriented techniques and Parallel DEVS. A chapter on hybrid systems shows how DEVS can be integrated with various differential equation solving techniques. For those interested in the latest developments in the field, Wainer & Mosterman (2011) provide a collection of recent DEVS research.

References

Barros, F. J. (1995). Dynamic structure discrete event system specification: A new formalism for dynamic structure modeling and simulation. In *Proceedings of the 27th conference on Winter simulation (WSC)* (pp. 781-785).

Castro R., Kofman E., & Wainer G. A. (2008). A formal framework for stochastic DEVS modeling and simulation. In *Proceedings of the 2008 Spring simulation multiconference (SpringSim)* (pp. 421-428).

Chow, A. C. H., & Zeigler, B. P. (1994). Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation (WSC)* (pp. 716–722).

Huang D., & Sarjoughian, H. (2004). Software and Simulation Modeling for Real-Time Software-Intensive Systems. In *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)* (pp. 196-203).

Nutaro J. J. (2011). *Building Software for Simulation: Theory and Algorithms with Applications in C++*. Hoboken, NJ, USA: John Wiley & Sons.

United Nations Environment Programme (2009). *Buildings and Climate Change: Summary for Decision Makers*.

Vangheluwe, H. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Proceedings of IEEE International Symposium on Computer-Aided Control System Design (CACSD)* (pp. 129–134).

Wainer, G. A., & Giambiasi, N. (2001). Timed Cell-DEVS: modelling and simulation of cell spaces. In *Discrete Event Modeling & Simulation: Enabling Future Technologies*. Springer-Verlag.

Wainer, G. A. (2009). *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL, USA: CRC Press.

Wainer, G. A., & Mosterman, P. J. (2011). *Discrete-Event Modeling and Simulation: Theory and Applications*. Boca Raton, FL, USA: CRC Press.

Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. New York: Wiley-Interscience.

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

Definitions

Atomic Model: An indivisible DEVS model specified with state transition functions, an output function, a time advance function, and sets of input values, output values, and states.

Closure Under Coupling: A property of the DEVS formalism which guarantees that the behavior of any coupled model can be captured by an atomic model specification.

Confluent Transition Function: A state transition function used in the Parallel DEVS variant to handle collisions between external and internal events.

Consistent: Describes a model specification that contradicts neither itself nor the conventions of the modeling formalism.

Coupled Model: A DEVS model composed of other DEVS models; a hierarchy is produced when coupled models are composed of other coupled models.

Discrete Event Simulation: A simulation in which time is repeatedly advanced by a variable, non-negative duration to the time of the next event.

Discrete Time Simulation: A simulation in which time is repeatedly advanced by a fixed time step.

External Transition Function: The state transition function invoked whenever an input is received.

Feedback Loop: A circular path in a coupled model formed by traversing couplings from their source submodels to their destination submodels.

Imminent: Describes a submodel that is scheduled to experience an internal event at least as soon as any other in the same coupled model.

Internal Transition Function: The state transition function invoked immediately after the output function.

Legitimate: Describes a model that, in the absence of inputs, is guaranteed to allow simulated time to advance towards infinity without stopping or converging.

Memoryless: Describes a model which has no state, and can therefore produce an output value that depends only on present information such as a just-received input.

Model Parameter: Represents a value that can be supplied to a model, but remains constant throughout a simulation.

Output Function: A function invoked to obtain an output value whenever the duration given by the time advance function elapses.

Port: A label assigned to a model to distinguish a particular type of input or output from other types of inputs or outputs.

Resultant: An atomic model that represents the behavior of a coupled model; closure under coupling guarantees that for every coupled model, a resultant exists.

Select Function: A function used to order the internal events of multiple imminent submodels; in the Parallel DEVS variant, the function is excluded to allow all imminent submodels to produce outputs simultaneously.

State: Any complete or partial, or raw or transformed record of a system's history.

State Variable: A variable used to represent part of a model's state.

Time Advance Function: A function invoked at the beginning of a simulation and after any state transition to give the duration that must elapse before the next internal event occurs.

Total State: Includes both the component of a system's state that remains constant between events, and the continuously changing time that has elapsed since the previous event.