Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A.A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

INTRODUCTION TO THE DISCRETE EVENT SYSTEM SPECIFICATION FORMALISM AND ITS APPLICATION FOR MODELING AND SIMULATING CYBER-PHYSICAL SYSTEMS

Gabriel A. Wainer

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

Rhys Goldstein
Azam Khan

Autodesk Research
210 King St. East
Toronto, ON, Canada

ABSTRACT

The Discrete Event System Specification (DEVS) formalism is a set of conventions for specifying discrete event simulation models. In this tutorial, we introduce the core concepts of DEVS. First, we introduce a set of informal requirements from which a formal specification is to be developed. Then, we present different modeling conventions at different levels of abstraction. The tutorial exploits the DEVS formalism's support for modular model design. The concepts are discussed with an example of cyber-physical systems modeling and implementation, which can be used to understand the main concepts of the formalism.

1 INTRODUCTION

Discrete Event System Specification (DEVS), a formalism for specifying discrete event simulation models, was first 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 tutorial is organized around a particular example: the simulation of a cyber-physical system, in this case, represented by a line-tracking robotic system. We develop this example from a set of informal requirements to a complete formal specification. The tutorial is based on a book chapter written by the authors, which can be found at (Goldstein et al. 2013).

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. 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 we will present a procedure like that in Figure 1, but suitable for 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.

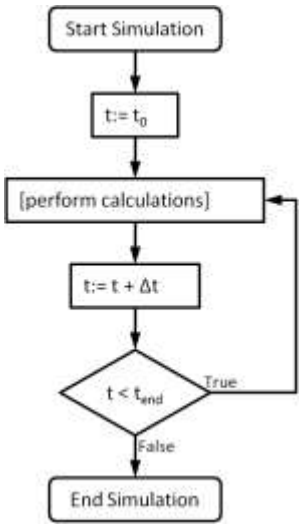

Figure 1: Simulation algorithm.

Here we show the DEVS formalism's support for modular model design, and we show an application in the field of cyber-physical systems (a simple line tracking robot, which will be used as an example to show the main concepts and ideas of the formalism, and its application to a concrete problem simple to understand). 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. We then discuss the definition of a simulation environment to develop such applications, and present an actual application of the formal models discussed in the article.

2 OVERVIEW OF THE DEVS FORMALISM

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 model an Infrared Sensor (IR) connected to a simple line-tracking robot designed to follow a track identified by a dark line, and to get back on track if the trail is not detected. The IR_Sensor 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:

$$\langle X, Y, \lambda \rangle \quad \text{is the structure of a Formalism A model specification}$$
$$X \quad \text{is the set of input values}$$
$$Y \quad \text{is the set of output values}$$
$$\lambda: X \to Y \quad \text{is the output function}$$

Following, we can see the IR_Sensor model specified using Formalism A:

$$IR\_Sensor_A = \langle X, Y, \lambda \rangle$$
$$X = \{(\text{"color}_{\text{in}}\text{"}, color) |\ color \in \{\text{"}blue\text{"}, \text{"}black\text{"}, \text{"}yellow\text{"}, \text{"}white\text{"}\}\}$$
$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) \big|\ signal \in \{\text{"}On\text{"}, \text{"}Off\text{"}\}\}$$

$$\lambda\big(("\text{color}_{\text{in}}", action)\big) = \big("\text{signal}_{\text{out}}", signal\big)$$
$$(color \in \{"\text{blue}", "\text{black}"\}) \Longrightarrow (signal = "\text{On}")$$
$$(color \in \{"\text{white}", "\text{yellow}"\}) \Longrightarrow (signal = "\text{Off}")$$

There are a couple of things in the specification worth noting. First, we have defined an input port "color_{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 "blue" and "black" input colors to the output signal "On" (i.e., when we detect we are on the track, we must turn on the robot's motors), and the other maps both "white" and "yellow" to "Off" (we are off-track so we turn off the motors). There is a subtle problem with this specification. If a "black" action is received, followed by "blue", the model will output two consecutive "On" signals. If it receives two consecutive "white" actions, it will send two consecutive "Off" signals. We want the signals "On" and "Off" 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*** remain constant throughout a simulation.

If we want to simulate the IR_Sensor model with Formalism A, we must define the simulation procedure associated with the formalism. Illustrated in Figure 2, the procedure is simple. 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 ∞.

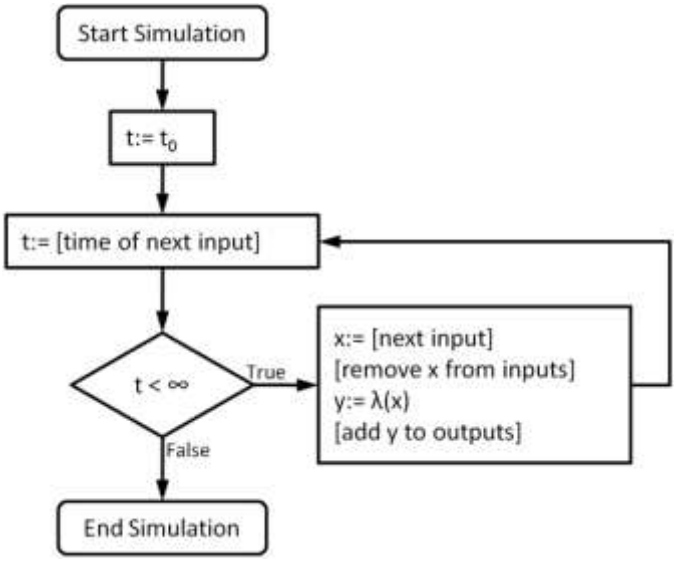

Figure 2: Simulation procedure for Formalism A.

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 *IR_Sensor* model). To address this issue, let us propose a more complex formalism, called Formalism B, which is similar to Formalism A, but models now have state. The state of a model remains constant between events, but may change during any event:

$\langle X, Y, S, \delta, \lambda \rangle$ 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 one. 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 IR_Sensor model specification below is lengthier than the previous, but we have introduced behavior that we could not previously describe.

$$IR\_Sensor_B = \langle X, Y, S, \delta, \lambda \rangle$$

$$X = \{(\text{"color}_{\text{in}}\text{"}, color) \mid color \in \{\text{blue, black, yellow, white}\}\}$$

$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) \mid signal \in \text{"}On\text{"}, \text{"}Off\text{"}\}$$

$$S = \{\text{ "On", "Off" }\}$$

$$\delta(signal, (\text{"color}_{\text{in}}\text{"}, color)) = signal'$$
$$color \in \{\text{"blue", "black"}\} \Longrightarrow (signal' = \text{"On"})$$
$$color \in \{\text{"white", "yellow"}\} \Longrightarrow (signal' = \text{"Off"})$$

$$\lambda(signal, (\text{"color}_{\text{in}}\text{"}, color)) = (\text{"signal}_{\text{out}}\text{"}, signal')$$
$$\begin{pmatrix} color \in \{\text{"white", "yellow"}\} \\ signal = \text{"On"} \end{pmatrix} \Longrightarrow (signal' = \text{"Off"})$$
$$\begin{pmatrix} action \in \{\text{"blue", "black"}\} \\ signal = \text{"Off"} \end{pmatrix} \Longrightarrow (signal' = \text{"On"})$$
$$\begin{bmatrix} \text{above conditions} \\ \text{are all false} \end{bmatrix} \Longrightarrow (y = \emptyset)$$

Observe that the transition function δ records the previous output, either "On" or "Off", in the state variable *signal*. Likewise, note the changes to the output function λ, which now depend on *signal*. We only output "On" if *signal* was previously "Off" and vice-versa. Also, if neither of the first two conditions is met, we output $\emptyset$. Figure 3 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.

Formalism B appears well suited to the IR_Sensor model. However, our modeling requirements are about to get steeper. The sensor is connected to a Motor model. After receiving an "On" input signal, the motor will turn off after a time of Δt_{Speed}, to model the motor's speed. So after Δt_{Speed} elapses, the Motor 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, the state remains constant between events. The problem is that the state of a real-world system may change continuously over time. Take our line-tracking robot, for example. After we give the instruction to the motor to turn on, the system is in such a state that it will turn off after a time of Δt_{Speed}. One infinitesimal time dt later, the robotic system is in an entirely new state "the motor will turn off after time $\Delta t_{Speed} - dt$". And the system passes through an *infinite* number of states like this one before Δt_{Speed} elapses and the motor turns off. Fortunately, this can be captured by a single variable: Δt_e, which represents the time elapsed since the previous event. If we know that Δt_e time units have elapsed the last change, we know that the motor will turn off after a time of $\Delta t_{Speed} - \Delta t_e$.

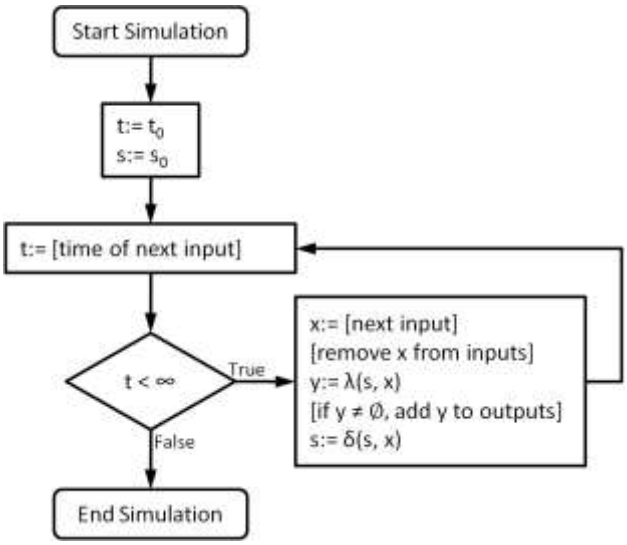

Figure 3: Formalism B Simulation Procedure.

Having acknowledged the importance of the elapsed time, we now have a means to represent two types of state: our original type of state, *s*, which remains constant between events (which we will still call *s* as "the state"), and the ***total state***, (s, Δt_e), which reflects the continuously changing state of a real-world system. 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.

2.1 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. 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

$Q = \left\{ (s, \Delta t_e) \middle| \begin{matrix} s \in S \\ 0 \le \Delta t_e \le ta(s) \end{matrix} \right\}$ 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

$T = \{\Delta t_{int} | 0 \le \Delta t_{int} \le \infty\}$ 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. At what simulated time, exactly, are λ and δ_{int} invoked? The answer is provided by the ***time advance function*** *ta*. 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 use s but not the elapsed time Δt_e. Passing Δt_e into λ or δ_{int} is unnecessary, as they are evaluated when Δt_e is equal to $ta(s)$. Thus, if the total state is needed during an internal event, one simply evaluates the time advance function and obtains the elapsed time. Based on these, we can now show a discrete event simulation flowchart based on DEVS, as seen in Figure 4.

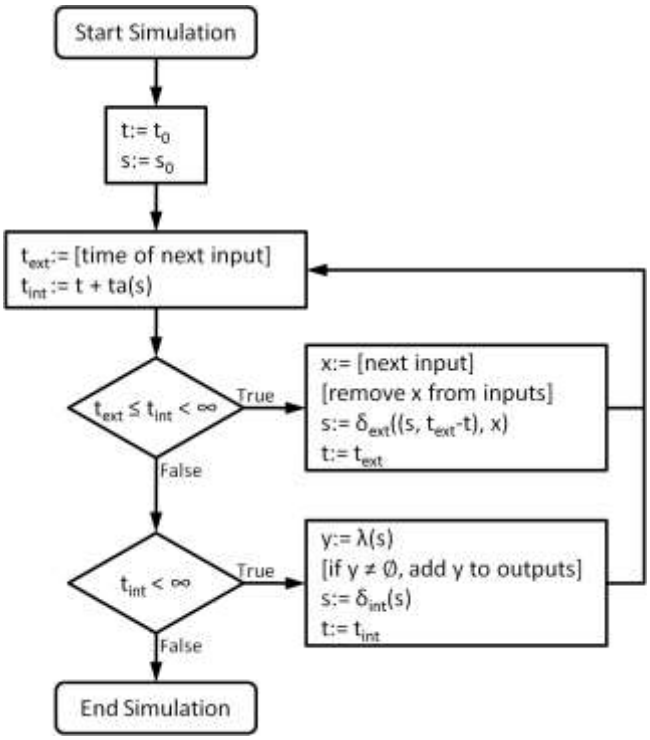

Figure 4: Discrete event simulation procedure using DEVS.

We 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 $ta(s)$? External events take priority over internal events.

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

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

$$X = \{(\text{"color}_{\text{in}}\text{"}, color) \mid color \in \{\text{blue, black, yellow, white}\}\}$$

$$Y = \{(\text{"signal}_{\text{out}}\text{"}, signal) \mid signal \in \text{"}On\text{"}, \text{"}Off\text{"}\}$$

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

$$\delta_{ext}\left(((signal, sent), \Delta t_e), (\text{"color}_{\text{in}}\text{"}, color)\right) = (signal', sent')$$

$$\begin{pmatrix} color \in \{\text{"blue", "black"}\} \\ signal = \text{"Off"} \end{pmatrix} \Rightarrow \begin{pmatrix} signal' = \text{"On"} \\ sent' = \bot \end{pmatrix}$$

$$\begin{pmatrix} color \in \{\text{"white", "yellow"}\} \\ signal = \text{"On"} \end{pmatrix} \Rightarrow \begin{pmatrix} signal' = \text{"Off"} \\ sent' = \bot \end{pmatrix}$$

$$\begin{bmatrix}\text{above conditions}\\ \text{are all false}\end{bmatrix} \Rightarrow \begin{pmatrix} signal' = signal \\ sent' = \top \end{pmatrix}$$
$$\delta_{int}\big((signal, \bot)\big) = (signal, \top)$$
$$\lambda\big((signal, \bot)\big) = \big(\text{"signal}_{\text{out}}\text{"}, signal\big)$$
$$ta\big((signal, sent)\big) = \Delta t_{int}$$
$$\neg sent \Longrightarrow (\Delta t_{int} = 0) \qquad sent \Longrightarrow (\Delta t_{int} = \infty)$$

The new δ_{ext} looks like the output function in Formalism B, but we store the state "On/Off" in a state variable to be output at a later stage. Another difference is that there is no longer a need to output $\emptyset$. Instead, we make use of the new state variable *sent*, which is either true ($\top$) or false ($\bot$), to avoid unwanted outputs: if an input is received, δ_{ext} updates the state and the time advance function will be evaluated. In the case that δ_{ext} changes the signal from "On" to "Off" (or vice-versa), *sent* is assigned $\bot$ and $ta(s)$ is 0, causing an instantaneous internal event. Once the output value $\lambda(s)$ is sent, δ_{int} changes *sent* to $\top$. 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]"). According to the specification, *sent* must end up $\top$, and thus $ta(s)$ will yield ∞. The $ta(s) = \infty$ prevents λ from being evaluated at all.

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. 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).

Whenever we specify a model using DEVS, we ought to ensure that 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. 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.

2.2 DEVS Coupled Models

To complete a specification we need to link the atomic models together as submodels of a DEVS coupled model. The conventions for doing this are given 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

$M: D \rightarrow \mathcal{M}$ 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
$Select: 2^D \rightarrow D$ 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 Robotic model an input port to start/end the activity, which will track a line. We will also define two output ports, one for the robot's actions and another one for the robot speed. Below is the specification of the Robotic model, with the exception of the tie-breaking *Select* function.

$$LTRobot(\Delta t_{speed}) = \langle X, Y, D, M, EIC, EOC, IC, Select \rangle$$
$$X = \{(\text{"command}_{\text{in}}\text{"}, c) \mid c \in \{start, end, sensor\_\}\}$$
$$Y = Y_{speed} \cup Y_{action}$$
$$Y_{action} = \{(\text{"action}_{\text{out}}\text{"}, action) | action \in \{straight, left, right, stop\}\}$$
$$Y_{level} = \{(\text{"speed}_{\text{out}}\text{"}, s) | s \in \{0 - 5\ km/h\}\}$$
$$D = \{\text{"IR\_Sensor"}, \text{"Motor"}, \text{"Control\_U"}\}$$
$$M(d) = m$$
$$(d = \text{"IR\_Sensor"}) \Longrightarrow (m = IR\_Sensor)$$
$$(d = \text{"Motor"}) \Longrightarrow \left(m = DCMotor(\Delta t_{speed})\right)$$
$$(d = \text{"Control\_U"}) \Longrightarrow (m = PID\_Controller)$$
$$EIC = \{((\text{"LTRobot"}, \text{"command}_{\text{in}}\text{"}), (\text{"Control\_U"}, \text{"start\_stop}_{\text{in}}\text{"}))\}$$
$$EOC = \left\{ \begin{matrix} ((\text{"Motor"}, \text{"speed}_{\text{out}}\text{"}), (\text{"}LTRobot\text{"}, \text{"speed}_{\text{out}}\text{"})), \\ ((\text{"Control\_U"}, \text{"action}_{\text{out}}\text{"}), (\text{"}LTRobot\text{"}, \text{"action}_{\text{out}}\text{"})) \end{matrix} \right\}$$
$$IC = \left\{ \begin{matrix} ((\text{"Control\_U"}, \text{"action}_{\text{out}}\text{"}), (\text{"Motor"}, \text{"action}_{\text{in}}\text{"})), \\ \left((\text{"IR\_Sensor"}, \text{"signal}_{\text{out}}\text{"}), (\text{"Control\_U"}, \text{"signal}_{\text{in}}\text{"})\right), \\ ((\text{"Motor"}, \text{"speed}_{\text{out}}\text{"}), (\text{"Control\_U"}, \text{"speed}_{\text{in}}\text{"})) \end{matrix} \right\}$$

The set *D* contains a unique ID for each submodel. Each coupling between ports takes the form (([source ID], [output port]), ([destination ID], [input port])). The specification shows that the LT_Robot's input port, "command$_{in}$", is connected to the "start_stop$_{in}$" input port of the "Control_U" submodel. The port names can match or be different, like in this case. The relationship is represented by (("LT_Robot","command$_{in}$"), ("Control_U", "start_stop$_{in}$")) in the set *EIC*.

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 Robotic model to the individual submodels. Here we are treating *M* as a function that maps an ID *d* to the corresponding DEVS submodel $M(d) = \langle X_d, Y_d, \ldots \rangle$.

Earlier we presented the simulation procedure associated with DEVS atomic models. 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.

$\langle X, Y, D, M, EIC, EOC, IC, Select \rangle$ 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 eight variables that compose a coupled model can be mapped into the seven 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 "$\text{command}_{\text{in}}$" input received by the Robotic model will get redirected to the "$\text{start\_stop}_{\text{in}}$" port of the Control_U model. The Control_U 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 IR_Sensor model triggers an internal event, a "$\text{signal}_{\text{out}}$" output will be sent to the Motor model. The Motor 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 ∅. If the Motor model sends an output, the Robotic model sends the same output as well. But, if the IR_Sensor model sends an output, the Robotic model outputs ∅. When an event of any kind occurs in a coupled model, the elapsed time associated with every submodel is updated.

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 Robotic model:

$$
\begin{aligned}
&Select(D_{imm}) = d_s \\
&\quad (\text{"Motor"} \in D_{imm}) \Longrightarrow (d_s = \text{"Motor"}) \\
&\quad \begin{pmatrix} \text{"IR\_Sensor"} \in D_{imm} \\ \text{"Motor"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"IR\_Sensor"}) \\
&\quad \begin{pmatrix} \text{"Control\_U"} \in D_{imm} \\ \text{"IR\_Sensor"} \notin D_{imm} \\ \text{"Motor"} \notin D_{imm} \end{pmatrix} \Longrightarrow (d_s = \text{"Control\_U"})
\end{aligned}
$$

Suppose the IR_Sensor and Motor submodels are both imminent. According to *Select*, the Motor model experiences the internal event first. By the time the IR_Sensor model triggers an internal event and signals the Motor model, the Motor model is in a new state and is no longer imminent. In a similar fashion, *Select* prevents the Control_U model from sending actions to an imminent IR_Sensor model.

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. A feedback loop in a coupled model is any circular path formed by traversing couplings from their source submodels to their destination submodels. 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. If the time required for a sequence of events to propagate around a feedback loop is either equal 0 or their sum converges on 0, the model is not legitimate.

3 CASE STUDY: A LINE TRACKING ROBOT

In this section, we discuss the simulation of a model based on the specifications above using the CDBoost simulator (Vicino et al. 2015), a DEVS simulator built as an extension of the CD++ simulator (Wainer 2009). At the time of the publication of Vicino et al. (2015), the CDBoost was the fastest existing DEVS simulator. The CDBoost simulator provides a library of DEVS simulation engines based on the ideas discussed in Section 2. An embedded version of the tool, called E-CDBoost (Niyonkuru and Wainer 2016) can be used to run the models in real-time mode, using a variety of embedded devices. In this section, we show how to use it to build an actual cyber-physical system, in this case, a simple line tracking robotic application like the one discussed in the previous section.

There are model classes that allow for defining the DEVS models, and execution classes that implement the simulation algorithms. Utility classes provide useful functions such as time classes, message classes, input streams for external events and a future event list. Model classes contain three main classes: *Model,* which offers a common interface to atomic and coupled models, *PDEVSAtomic*, which can be extended to implement user-defined atomic models, and *PDEVSCoupled,* which provides an interface to specify the structure of a model. These are the classes the users define to build their models and run simulations.

Using these services, we will show now how to define our line-tracking robot. The controller, which was not defined in detail in Section 2, considers a medium percentage of reflected light as a detected path and initiates the robot to move forward. When the robot goes off track, i.e., does not sense a path trail, it stops, turns slightly, and then tries to detect a trail again. The robot also receives manual signals to start and stop. In terms of components, the sensor unit contains input devices. The sensor controller activates or stops the light sensor, receives the sensor readings, and sends messages to the motor controller, specifying whether the robot is on track, off track, or has reached the destination. The controller also receives on/off track and stop signals from the sensor, and it sends appropriate commands to the motors. Figure 5 illustrates a DEVS Graph representing the sensor controller's behavior.

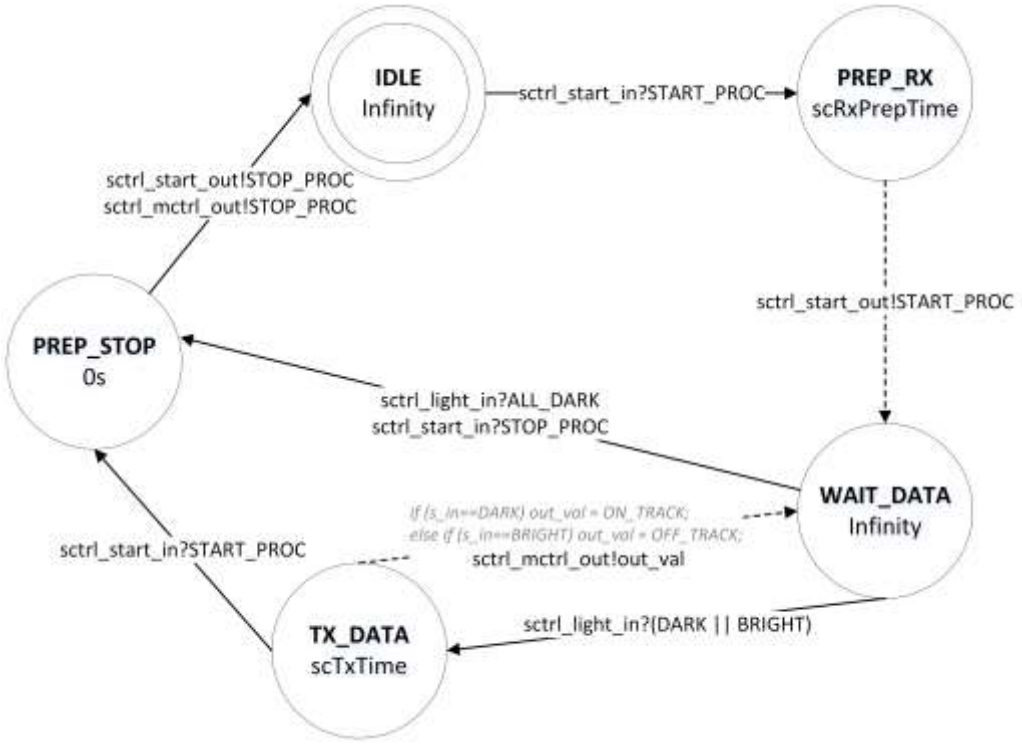

Figure 5: Sensor controller state diagram.

This graphical representation is an extended version of the notation used in Section 2, which we can then translate to CDBoost. As we can see, the Sensor Controller is IDLE until a *start* command is issued. Then, an external transition is triggered and the Sensor Controller state changes to PREP_RX. At this point, it waits for *ta=scRxPrepTime*, after which a 'start' output is sent to the Light Sensor and an internal transition changes state to WAIT_DATA. It waits in this state until it receives a signal from the Light Sensor. If the signal indicates that the robot reached the destination (ALL_DARK), the external transition causes a switch to PREP_STOP, where it will immediately send a stop signal to the Light Sensor and the Movement Controller, and it will transition back to IDLE. However, if the signal is different, the Sensor Controller will go to TX_DATA, will wait for *ta=scTxTime*, after which it will send an output to the Movement Controller indicating whether the robot is on track or not. If the Sensor Controller receives a manual stop signal (STOP_PROC), it will transition to the PREP_STOP to stop all activities.

In order to implement atomic models in CDBoost, we extend the basic model class providing state transition and output functions. The code in Figure 6 shows an example for the sensor controller functions. We can see that it includes the state transition and output functions that correspond to the original DEVS specification. A message is constructed using the port and the value to be sent. The TIME parameter returned by the time advance function is defined using real time units.

```
void internal() noexcept {
    switch (_state){
        case PREP_STOP:  _state = IDLE;   _next = infinity;  break;
        case PREP_RX:
        case TX_DATA: _state = WAIT_DATA; _next = infinity;  break;       }
}

TIME advance() const noexcept { return _next; }
     /* @return Time until next internal event. */

std::vector<MSG> out() const noexcept {   ...
 switch (_state){
   case PREP_STOP:      //Send stop through IR_Sensor _start_out and mctrl
      _outputMessage1 = MSG(portName[IR_Sensor _start_out], STOP_PROC);
         _outputMessage2 = MSG(portName[IR_Sensor _mctrl_out], STOP_PROC);
         std::vector<MSG>{_outputMessage1, _outputMessage2};
   case PREP_RX: //Send Start through IR_Sensor _start_out
         _outputMessage1 = MSG(portName[IR_Sensor _start_out], START_PROC);
         return std::vector<MSG>{_outputMessage1};
  case TX_DATA:    //Send on/off track signals IR_Sensor _mctrl_out
     int output_val;
         if(sensor_input == DARK) output_val = ON_TRACK;
                else if (sensor_input == BRIGHT) output_val = OFF_TRACK;
         _outputMessage1 = MSG(portName[IR_Sensor _mctrl_out], output_val);
             return std::vector<MSG>{_outputMessage1};
  };
   return std::vector<MSG>{}; //Default: empty output      }
```

Figure 6: Atomic model definition.

To implement the coupled model, we use the syntax shown in Figure 7.

```
auto IR_Sensor  = make_atomic_ptr <IR_Sensor<Time, Message>>(); // Atomic models definition
auto Control_U= make_atomic_ptr <Control_U<Time, Message>>();
shared_ptr< coupled<Time, Message>> LTRobot( new coupled<Time, Message>{{IR_Sensor ,mctrl},
              {IR_Sensor }, {{IR_Sensor ,mctrl}}, {mctrl}});    //Coupled model definition
```

Figure 7: Coupled model definition.

The sensor model (IR_Sensor at line 1) and control unit (Control_U at line 2) are the two of the components of the coupled model. If we want to test them individually, the model on line 3 can be used; it includes its components ({IR_Sensor ,mctrl}), then its EIC (signals from hardware components; IR_Sensor is connected to the light sensor and push button), its IC (IR_Sensor is connected to Control_U internally), and finally its EOC (components sending output signal to hardware: Control_U to the two motors).

We will illustrate the execution mechanism using trace logs collected during the execution of the line tracking robot. Two examples are provided to illustrate internal execution mechanism.

```
DRIVER: INPUT MESSAGE     Time: 02:517:459
 Port: start_in Value: 10
 - advance_execution()::LTRobot; advance_execution()::IR_Sensor
   model->external() model->advance(): 00:040:000
 - collect_outputs()::LTRobot; advance_execution()::LTRobot
 - collect_outputs()::IR_Sensor ; model->out()
 - advance_execution()::IR_Sensor    model->internal() model->advance(): ...
 - advance_execution()::Control_U    model->external() model->advance(): ...
DRIVER: INPUT MESSAGE  Time: 02:600:697
 Port: light_in Value: 1
- advance_execution()::LTRobot; advance_execution()::IR_Sensor
   model->external() model->advance(): 00:040:000
- collect_outputs()::LTRobot; advance_execution()::LTRobot
 - collect_outputs()::IR_Sensor          model->out()
 - advance_execution()::IR_Sensor    model->internal() model->advance(): ...
 - advance_execution()::Control_U
   model->external() model->advance(): 00:040:000
 - collect_outputs()::LTRobot  collect_outputs()::Control_U    model->out()
DRIVER: OUTPUT MESSAGE  Time: 02:680:850
 Port: motor Value: 1
```

Figure 8: Simulation results. Internal transition execution**.**

Figure 8 shows a sequence that follows a *start* message at time 02:517:459. An input message with value 10 (turn on) triggers a call to the external function of the sensor model. An input message indicating a line detection is then sent and causes the sensor and controller external functions to be called. Two outputs are generated, commanding the motors to go forward (Value 1 sent to the motor).

Figure 9 shows the case corresponding to a manual stop that causes stop commands (0 sent to the motor).

Once the tests are done, the controller model is deployed onto a robotic device built using a Nucleo board to autonomously control the robot. A number of videos showing the result on the target platform are available at http://www.youtube.com/arslab. Following the model, the robot tracks the dark lines on the floor, and whenever it detects a junction it takes a right turn, as seen in Figure 10. If it does not detect a line it will take a turn and search for the black line.

```
DRIVER: INPUT MESSAGE  Time: 00:02:10:403:002
 Port: start_in Value: 11
 - advance_execution()::LTRobot
 - advance_execution()::IR_Sensor
   model->external() model->advance(): 00:000:000
 - collect_outputs()::LTRobot
 - advance_execution()::LTRobot
 - collect_outputs()::IR_Sensor     model->out()
 - advance_execution()::IR_Sensor   model->internal() model->advance(): ...
 - advance_execution()::mctrl
   model->external() model->advance(): 00:000:000
 - collect_outputs()::LTRobot
 - collect_outputs()::mctrl  model->out()
DRIVER: OUTPUT MESSAGE  Time: 00:02:10:403:559
 Port: motor Value: 0
```

Figure 9: Simulation results.

Figure 10: Line tracking model real-time execution.

4 SUMMARY

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 robotic 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).

DEVS users should familiarize themselves with several variants of the formalism. One of these variants is Parallel DEVS. Another variant is Cell-DEVS, which applies DEVS to models composed of an

array of cells (Wainer and 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 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, 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 and 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 1995 Winter Simulation Conference*, edited by C. Alexopoulos et al., 781-785. Piscataway, New Jersey: IEEE.

Castro R., E. Kofman, and G. A. Wainer. 2008. "A Formal Framework for Stochastic DEVS Modeling and Simulation". In *Proceedings of the 2008 Spring Simulation Multiconference (SpringSim),* 421-428. San Diego, CA, USA: Society for Computer Simulation International.

Chow, A. C. H. and B. P. Zeigler. 1994. "Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew et al., 716–722. Piscataway, New Jersey: IEEE.

Goldstein, R., G. A. Wainer, and A. Khan. 2013. "The DEVS Formalism". In *Formal Languages for Computer Simulation: Transdisciplinary Models and Applications*, edited by P. Fonseca i Casas, 62-102. Hershey, Pennsylvania: IGI Global.

Niyonkuru, D. and G. A. Wainer. 2016. "A Kernel for Embedded Systems Development and Simulation Using the Boost Library". In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation*, art 3. San Diego, CA, USA: Society for Computer Simulation International.

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

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)*, 129–134. Piscataway, New Jersey: IEEE.

Vicino, D., D. Niyonkuru, G. Wainer, and O. Dalle. 2015. "Sequential PDEVS Architecture". In *Proceedings of the 2015 Symposium on Theory of Modeling & Simulation,* 165-172. San Diego, CA, USA: Society for Computer Simulation International.

Wainer, G. A. and N. Giambiasi. 2001. "Timed Cell-DEVS: Modelling and Simulation of Cell Spaces". In *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*, edited by H.S. Sarjoughian and F.E. Cellier, 187-214. New York, NY, USA: Springer.

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

Wainer, G. A. and P. J. Mosterman. 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, NY, USA: Wiley-Interscience.

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

AUTHOR BIOGRAPHIES

GABRIEL A. WAINER (FSCS, SMIEEE), received the M.Sc. (1993) at the University of Buenos Aires, Argentina, and the Ph.D. (1998, with highest honors) at the Université d'Aix-Marseille III, France. In 2000 he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now Full Professor and Associate Chair for Graduate Studies. He is the author of three books and over 350 research articles; he edited four other books, and helped organizing numerous conferences, including being one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools and SimAUD. Prof. Wainer was Vice-President Conferences and Vice-President Publications, and is a member of the Board of Directors of the SCS. Prof. Wainer is the Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE Computing in Science and Engineering, Wireless Networks (Elsevier), Journal of Defense Modeling and Simulation (SCS). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005, 2014), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award (2013), the SCS Distinguished Professional Award (2013), and the SCS Distinguished Service Award (2015). He is a Fellow of SCS. He was the Program Chair of WinterSim 2017. His e-mail address is gwainer@sce.carleton.ca.

RHYS GOLDSTEIN is a simulation expert in the Complex Systems group at Autodesk Research. His work straddles the disciplines of simulation theory and architectural design, and he has contributed as a researcher and conference organizer in both areas. Rhys served as the 2014 Co-Chair and 2015 Program Chair of the Simulation for Architecture and Urban Design (SimAUD) Symposium, as well as the 2017 Program Co-Chair of the Theory of Modeling and Simulation (TMS/DEVS) Symposium. He is currently an Associate Editor of the journal SIMULATION, and lead developer of the SyDEVS open source C++ library for systems modeling. His e-mail address is rhys.goldstein@autodesk.com.

AZAM KHAN is Director, Complex Systems Research at Autodesk. He is the Founder of the Parametric Human Project Consortium, SimAUD: the Symposium on Simulation for Architecture and Urban Design, and the CHI Sustainability Community. He is also a Founding Member of the International Society for Human Simulation and has been the Velux Guest Professor at The Royal Danish Academy of Fine Arts, School of Architecture, at the Center for IT and Architecture (CITA) in Copenhagen, Denmark. Azam has published over 50 articles in simulation, human-computer interaction, architectural design, sensor networks, and sustainability. His e-mail address is azam.khan@autodesk.com.