Practical Aspects of the DesignDEVS Simulation Environment

Rhys Goldstein, Simon Breslav, Azam Khan

Abstract

DesignDEVS is a simulation development environment based on the Discrete Event System Specification (DEVS) formalism. This paper provides an in-depth overview of the software while focusing on the practical considerations influencing its design. Practitioners who stand to benefit from systems engineering will approach formalism-based simulation tools with little knowledge of the underlying theory. It is therefore important that theoretical principles, such as the separation of model and simulator, be emphasized by the user interface. Other practical aspects of DesignDEVS include the simplicity of atomic model code, a focus on coupling for collaboration purposes, the enforcement of essential modeling constraints, and a reliance on best practices in cases where strict enforcement might inconvenience users. In DesignDEVS, an issue we refer to as the Insidious Pointer Problem is aggressively tackled through run-time error handling. By contrast, the separation of output values from state transitions is left as a best practice for the sake of user convenience. The design decisions explained in this paper are relevant to developers of other formalism-based tools seeking widespread adoption of scalable modeling and simulation practices.

Keywords

Simulation environments, modeling formalisms, discrete-event simulation, error handling, encapsulation, best practices

1. Introduction

An overarching goal of the field of modeling and simulation is to encourage experts from a wide range of disciplines to build communities of practice using scalable methods that will ultimately give rise to collaboratively authored predictive models of the most complex natural and/or artificial systems that humans encounter and/or design. Yet practitioners tend to stick with familiar programming techniques. They hesitate to explore alternative approaches that may or may not prove beneficial once all practical considerations are taken into account. The result is that systems engineering principles are rarely followed by the communities that have the most to gain from them. Ad-hoc simulators are typically embedded within model-specific code, causing redundancy and conflict when two or more models must be integrated. Moreover, models are often implemented with explicit references to one another, creating dependencies that discourage the testing of new combinations of models.

In seeking widespread adoption of scalable modeling and simulation practices, considerable attention must be paid to the practical aspects of theory-based simulation tools. This paper describes the practical considerations underlying DesignDEVS, a simulation development environment based on the Discrete Event System Specification (DEVS) formalism[1] and intended for collaborative multi-disciplinary design and engineering modeling. As outlined in previous work[2], DesignDEVS reinforces its users' understanding of theoretical principles such as model-simulator separation and delayed binding of models. The embedding of the Lua[3] scripting language allows the environment to be distributed in a small and self-contained package, and eliminates the step of generating executable code from users' models. In this paper we provide examples which explain the design decisions behind the software. One example illustrates how DesignDEVS solves the Insidious Pointer Problem, a situation in which DEVS theory is compromised due to the passing of data references among models. Other examples reveal why certain theoretical principles are not strictly enforced, but merely encouraged through best practices.

Section 2 reviews DEVS theory and DEVS-based simulation environments, while the five subsequent sections each highlight a guideline concerning the design of such tools. Many of these design principles are relevant to various types of simulation environments, regardless of the underlying formalism. The guidelines and their re-

Autodesk Research, Canada

Corresponding author:
Rhys Goldstein, Autodesk Research, 210 King Street East, Suite 500
Toronto, Ontario, Canada M5A 1J7
Email: rhys.goldstein@autodesk.com

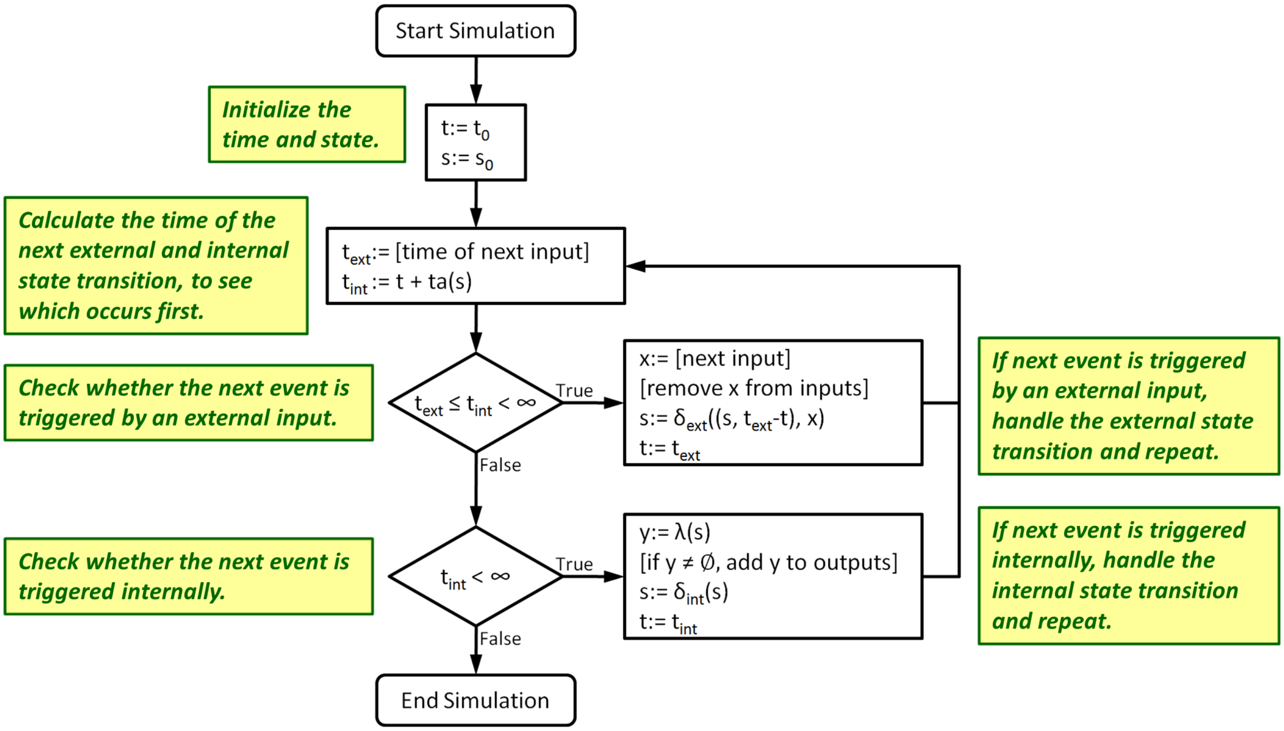

Start Simulation
Initialize the time and state.
t:= t0
s:= s0
Calculate the time of the next external and internal state transition, to see which occurs first.
text:= [time of next input]
tint := t + ta(s)
Check whether the next event is triggered by an external input.
text ≤ tint < ∞
True
False
x:= [next input]
[remove x from inputs]
s:= δext((s, text-t), x)
t:= text
If next event is triggered by an external input, handle the external state transition and repeat.
Check whether the next event is triggered internally.
tint < ∞
True
False
y:= λ(s)
[if y ≠ ∅, add y to outputs]
s:= δint(s)
t:= tint
If next event is triggered internally, handle the internal state transition and repeat.
End Simulation

Figure 1. The basic simulation procedure associated with the DEVS formalism.

lated sections are listed below, along with examples of how DesignDEVS addresses each issue:

1. Theoretical principles should be emphasized in the user interface, as this may be one of the users' only form of exposure to the theory. As illustrated in Section 3, the DesignDEVS user interface emphasizes the separation of model and simulator.

2. Unnecessary complexity in the atomic model code should kept to a minimum in order to avoid introducing additional barriers to the adoption of scalable methods. Section 4 explains how DesignDEVS minimizes boilerplate code using extensions to the Lua programming language.

3. Coupling is but one of many abstraction mechanisms available to the user, and the manner in which coupling is intended to be used will influence the design of the environment. As discussed in Section 5, a model hierarchy in DesignDEVS is intended to reflect a functional decomposition of a system. This intent goes hand-in-hand with a focus on collaboration, as well as the principle of delayed binding emphasized in the user interface.

4. Modeling constraints which uphold theoretical principles should be enforced, provided they do not place unbearable restrictions on users. Section 6 describes how metatables in Lua are used to detect and ultimately communicate a wide range of run-time errors.

5. In cases where the strict enforcement of convention would inconvenience users, it may be preferable to treat the convention as a best practice. A number of best practices for DesignDEVS are discussed in Section 7. The separation of output values from state transitions is notable in that a deliberate decision was made not to enforce the convention.

Past applications of DesignDEVS are presented in Section 8, while Section 9 concludes general advice on how formalism-based tools can address both practical considerations and the principles of the underlying theory.

2. Background and related work

Computer simulations are actively developed across a wide range of scientific and engineering domains. A variety of programming techniques are employed in these efforts, including acausal modeling as popularized by Modelica[4], and block-diagram editing as realized by Simulink[5] and Ptolemy II[6]. However many domain experts use traditional imperative programming as supported by C, C++, Java, Python, MATLAB, and even Fortran in many cases. The DEVS formalism offers a promising avenue for these practitioners to adopt scalable modeling and simulation practices. First, DEVS is among the most general of modeling formalisms. It has

DEVS-Based Environment	DEVS Variant	Modeling Language	Objective
PowerDEVS	Classic DEVS	C++	Promote DEVS-based quantized integrators to combine continuous and discrete models using block-diagram-like compositions similar to Simulink, Ptolemy II.
DEVS-Suite	Parallel DEVS	Java	Teach a systems approach to the modeling of computer networks and other systems, with animations of the simulation process superimposed on the model.
CoSMoS	Parallel DEVS	Java	Build upon the DEVS-Suite simulator with new visual modeling interfaces and a framework for categorizing and managing models and model families.
CD++ Builder	Classic DEVS + Cell-DEVS	C++	Reduce barriers for non-developer users with a state-diagram-like editor and other graphical modeling tools within an extensible Eclipse-based framework.
SimStudio	Classic DEVS	Java	Establish a multi-layer platform to support web-based collaborative authoring of simulation models.
VLE	Parallel DEVS + extensions	C++	Support heterogeneous model development and experimentation through a broad set of DEVS extensions and environment plug-ins.
DesignDEVS	Classic DEVS	Lua	Teach DEVS principles via modeling constraints enforced at run-time, while exploring best practices that account for scalability and user experience.

Table 1. A list of DEVS-based simulation environments indicating the underlying theory, programming language, and objective.

been shown that a multitude of other types of models can be formally mapped into DEVS models, though the reverse is not necessarily true[7]. Although the formalism is most often applied to artificial systems, progress has been made toward modeling the physics of motion in the context of particle systems[8] and fluid dynamics[9]. Second, DEVS lends itself well to the imperative style of programming familiar to virtually all scientists and engineers. Thus even when graphical features are incorporated into a DEVS-based simulation environment, the textual programming tasks that remain tend to build upon a user's preexisting knowledge.

At the core of the formalism are the seven elements of DEVS atomic models. Essentially any time-based simulation model can be specified by defining these elements.

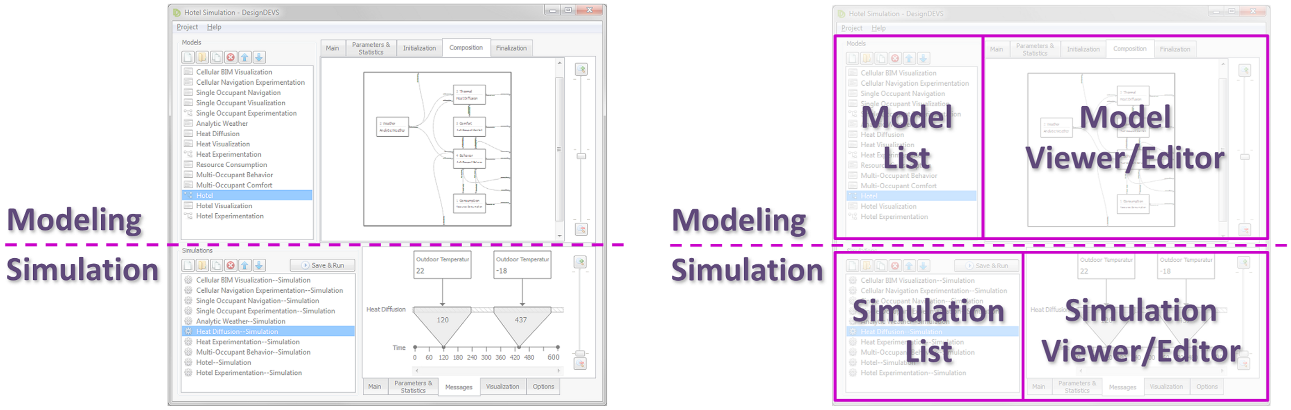
$$\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$

Here X, Y, and S are mathematical sets that restrict a model's allowable input, output, and state values, while δ_{ext}, δ_{int}, λ, and ta are mathematical functions invoked by a reusable simulator. A DEVS-based simulator must implement a procedure similar to that illustrated in Figure 1.

As seen in Figure 1, a simulation begins with the initialization of the current time t and state s. The next step is to obtain the time t_{ext} of the next input, and the time t_{int} of the next internally planned event. Note that t_{int} is derived from the model's time advance function ta. Depending on t_{int} and t_{ext}, one of three things may occur. First, an input x may be received. As shown in the first branch of the flow chart, the model's external transition function δ_{ext} is invoked to obtain a new state from the old state, the elapsed duration, and the input itself. The time t then advances and the process repeats. Second, an internal event may occur. As seen in the second branch of Figure 1, this entails the production of an output using the function λ, followed by a state change determined by the internal transition function δ_{int}. Again, time advances and the process repeats. The third possibility is that the simulation ends.

An atomic model can be assigned to a component in a coupled model. A coupled model can also be assigned to a component in an encompassing coupled model, forming a model hierarchy. This coupling mechanism helps one organize complex models. A comprehensive explanation of the formalism, including definitions of every mathematical element, an abstract simulator, and a formal description of coupled models, can be found in *Theory of Modeling and Simulation* by Zeigler et al.[1].

Numerous simulation development environments are available which provide both textual and graphical features for developing, debugging[10], and experimenting with simulation models. Among the simulation environments most dedicated to DEVS theory are those listed in Table 1: PowerDEVS[11], DEVS-Suite[12], CoSMoS[13], CD++ Builder[14], SimStudio[15], and VLE[16] (see Franceschini et al.[17] for a more comprehensive list). As indicated in the table, each tool is based on either the original 1970s version of the theory, Classic DEVS, or a 1990s variant called Parallel DEVS[18], though various



(a) A horizontal divider bar seperates modeling features from simulation features.

(b) Vertical divider bars separate model/simulation organizational features from model/simulation viewing and editing features.

Figure 2. The basic layout of the DesignDEVS user interface reflects (a) the DEVS principle of model-simulator separation, and (b) the organizational structure of users' projects.

extensions may be supported as well. Each environment handles atomic models implemented in a textual programming language, usually C++ or Java. All of these tools offer a node-link diagram editor for defining coupled models.

What distinguishes each simulation environment is a set of priorities and associated features, which we summarize with brief "Objective" statements in Table 1. Each tool will promote the concepts it most emphasizes, such as the quantization of state in the case of PowerDEVS, the management of models in the case of CoSMoS, or the integration of different types of graphical models in the case of CD++ Builder. A notable feature of CD++ Builder and similar tools is a state-diagram-like editor that provides an alternative to imperative programming for atomic models. Graphical models represent one strategy for promoting DEVS adoption. The DesignDEVS environment discussed in this paper follows a different strategy, attempting to take full advantage of domain experts' familiarity with procedural code.

Some environments use DEVS, but feature it less prominently than those in Table 1. The AToM^3 framework[19] exemplifies multi-paradigm modeling[20], where DEVS is regarded as a means of integrating models developed according to a diverse set of conventions. James II[21] also combines DEVS with other approaches. MS4 Me[22] is based on DEVS, but purposely minimizes its users' perceived exposure to systems theory by providing alternative modeling options such as sequence diagrams and natural language documents. DesignDEVS has more in common with the simulation environments in Table 1. As illustrated in Section 3, elements of DEVS theory are prominently exhibited in the user interface. However, as with similar tools, the emphasis on DEVS is not meant to downplay the importance of supporting modeling strategies seen as closer to the users' domains of expertise.

3. DesignDEVS user interface

DesignDEVS features a user interface consisting of four main quadrants separated by three adjustable divider bars. As shown in Figure 2a, the horizontal divider bar partitions the interface into an upper section dedicated to models, and a lower section dedicate to simulation runs based on those models. This structure was introduced with the intent of emphasizing a key principle of modeling formalisms in general: the separation of model and simulator[1]. All DEVS-based simulation environments feature this separation at a semantic level, but the visual partitioning of model and simulation elements reinforces the concept.

As indicated in Figure 2b, the two vertical divider bars separate project organization features from viewing and editing components. The organizational features on the left include a list of models (top left), a list of simulations (bottom left), and buttons to create, open, copy, delete, and reorder models or simulations. Reordering is an important operation in DesignDEVS, both from an aesthetic and functional perspective. For aesthetic reasons, a user may want to reorder models or simulations in order to cluster those that are similar to one another. From a functional perspective, it is sometimes necessary to reorder models in response to a DesignDEVS rule that any model A can only be assigned to a component of another model B if A appears above B in the model list. This rule prevents circular dependencies. In particular, it avoids a situation in which A and B are coupled models that include instances of one another, as both model hierarchies would then be infinitely deep.

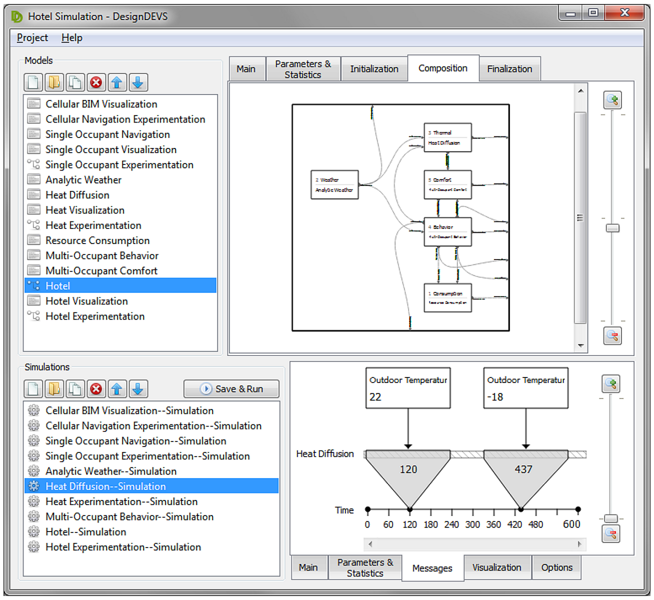
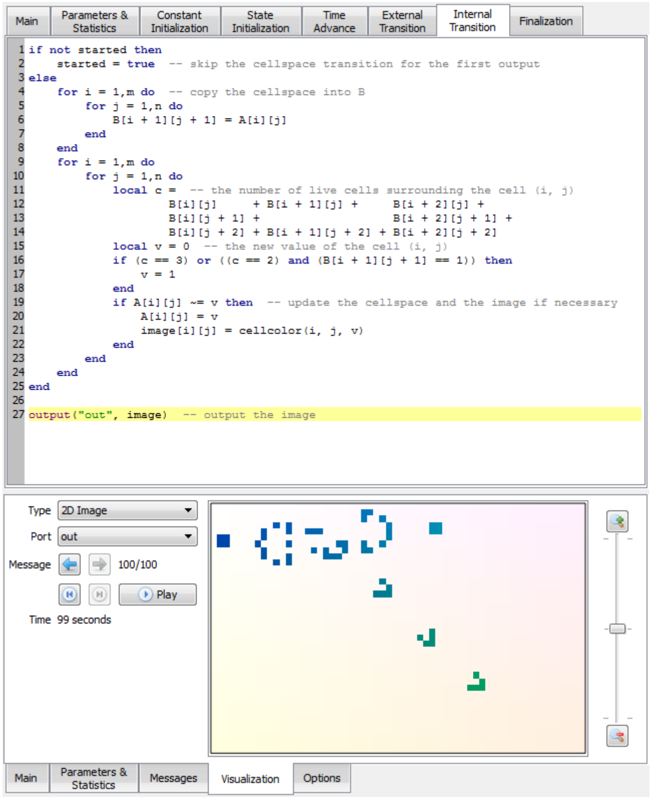


Figure 3. A screenshot of the DesignDEVS user interface.



```
if not started then
    started = true  -- skip the cellspace transition for the first output
else
    for i = 1,m do  -- copy the cellspace into B
        for j = 1,n do
            B[i + 1][j + 1] = A[i][j]
        end
    end
    for i = 1,m do
        for j = 1,n do
            local c =  -- the number of live cells surrounding the cell (i, j)
                    B[i][j]     + B[i + 1][j] +     B[i + 2][j] +
                    B[i][j + 1] +                   B[i + 2][j + 1] +
                    B[i][j + 2] + B[i + 1][j + 2] + B[i + 2][j + 2]
            local v = 0  -- the new value of the cell (i, j)
            if (c == 3) or ((c == 2) and (B[i + 1][j + 1] == 1)) then
                v = 1
            end
            if A[i][j] ~= v then  -- update the cellspace and the image if necessary
                A[i][j] = v
                image[i][j] = cellcolor(i, j, v)
            end
        end
    end
end

output("out", image)  -- output the image
```

Type 2D Image
Port out
Message 100/100
Time 99 seconds

Figure 4. A transition function populated with Lua code for a classic Game of Life model (top), and a 2D visualization of the results (bottom).

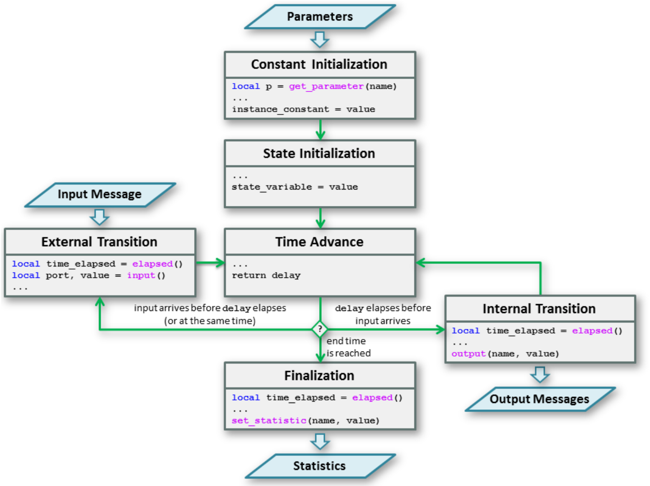
The viewing and editing components on the right of the vertical divider bars are grouped into tabs. In the modeling section, the tabs available depend on the type of model selected: atomic or coupled. Coupled models are associated with tabs named Main, Parameters & Statistics, Initialization, Composition, and Finalization. The Composition tab is selected in Figure 3, revealing the structure of the model. Atomic models are associated with tabs named Main, Parameters & Statistics, Constant Initialization, State Initialization, Time Advance, External Transition, Internal Transition, and Finalization. The Internal Transition tab is selected in Figure 4, revealing state transition code for an atomic model representing the classic Game of Life.

The simulations below the horizontal divider bar have a many-to-one relationship with the models. Although the modeling tabs depend on whether the selected model is atomic or coupled, the simulation viewing and editing tabs at the bottom right are the same regardless of whether the selected simulation is based on an atomic or coupled model. The fact that atomic and coupled models share a common interface is consistent with both the *closure under coupling* property of modeling and simulation theory[1] and the *Composite Design Pattern* familiar to software engineers[23].

The simulation tabs include Main, Parameters & Statistics, Messages, Visualization, and Options. The Messages tab, shown in at the bottom right of Figure 3, allows the user to edit input messages prior to executing the simulation run. To insert an input message, the user selects a point in simulated time by clicking on the axis at the bottom, chooses an input port from a pop-up menu, then types in a message value. Multiple input messages can be created for the same simulated time point. When the simulation is run, the user-defined input messages are received by the model at the specified points in simulated time, preserving the left-to-right order depicted in the timeline. The Messages tab then displays both the input messages and the resulting output messages. The Visualization tab, seen at the bottom of Figure 4, supports 2D animations of simulation results. To use the feature, one of the model's ports must periodically output a 2D array of {`red, green, blue`} values.

4. Atomic model development

In a typical DEVS-based simulation project, the majority of the user's code will be found in the atomic models. To encourage the adoption of the formalism, it is important that DEVS-based tools address the barriers facing domain experts who must learn to organize and write atomic model code. The first barrier is the new user's unfamiliarity with the various atomic model functions and the rules that govern how they are invoked by the simulator. DesignDEVS addresses this lack of familiarity by providing the simulation procedure diagram in Figure 5, which is accessible via the Help menu. The illustration is essentially an informal version of the flow chart in Figure 1. The DesignDEVS version uses labels



```
local p = get_parameter(name)
...
instance_constant = value
```

```
...
state_variable = value
```

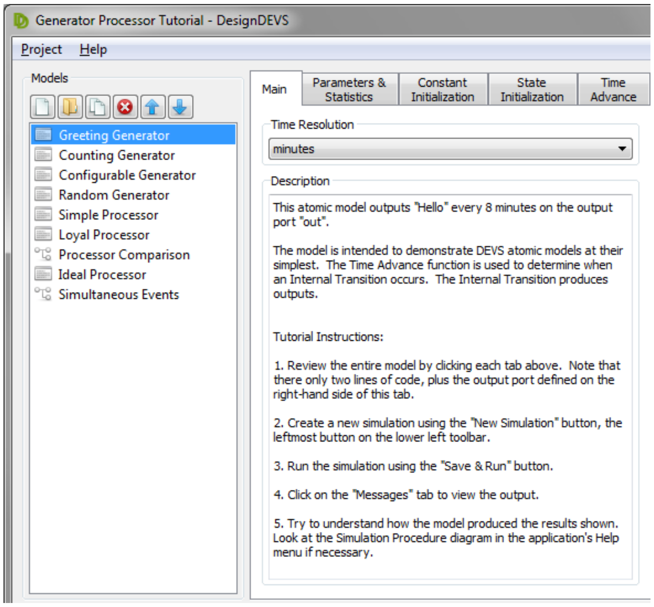
```
local time_elapsed = elapsed()
local port, value = input()
...
```

```
...
return delay
```

```
local time_elapsed = elapsed()
...
output(name, value)
```

```
local time_elapsed = elapsed()
...
set_statistic(name, value)
```

Figure 5. The DesignDEVS simulation procedure as illustrated in the Help menu, depicting the flow of execution from one tab to the next.



Time Resolution: minutes

Description

This atomic model outputs "Hello" every 8 minutes on the output port "out".

The model is intended to demonstrate DEVS atomic models at their simplest. The Time Advance function is used to determine when an Internal Transition occurs. The Internal Transition produces outputs.

Tutorial Instructions:

1. Review the entire model by clicking each tab above. Note that there only two lines of code, plus the output port defined on the right-hand side of this tab.

2. Create a new simulation using the "New Simulation" button, the leftmost button on the lower left toolbar.

3. Run the simulation using the "Save & Run" button.

4. Click on the "Messages" tab to view the output.

5. Try to understand how the model produced the results shown. Look at the Simulation Procedure diagram in the application's Help menu if necessary.

Figure 6. A model's description field can be used for a wide range of documentation purposes, including offering users a tutorial as in the Generator Processor Tutorial sample project.

that appear in the user interface (e.g. Constant Initialization, State Initialization, etc.), as well as snippets of code that may be useful in the corresponding tabs.

Another barrier facing domain experts is the inconvenience of reading and writing boilerplate code. DesignDEVS strives to support rapid prototyping by keeping atomic model focused, to the greatest extent possible, on the user's domain. To illustrate, Table 2 lists the source code for three basic models implemented in a mere 2, 4, or 18 lines. The 2-line Greeting Generator performs the same operation every time step. In this case, the repeated operation is the output of the string `"Hello"`, and the time step is 8 minutes of simulated time. The 4-line Counting Generator incorporates a state variable (`n`) and a state transition (`n = n + 1`). The 18-line Ideal Generator demonstrates input messages. In this model, an input message is passed to the output as quickly as possible.

To help new users, the models in Table 2 and several others are included in a Generator Processor Tutorial project packaged with the software. This sample project features a list of models, and each model has step-by-step tutorial instructions written into its Description field as shown in Figure 6. To promote documentation in general, all models and simulations have this field.

The minimization of unnecessary boilerplate code is enabled in large part by the embedding of Lua as the primary programming language for model development. Lua is a high-performing yet lightweight scripting language that has achieved popularity within the computer game industry. Among the features that make Lua simple is the fact that it provides only one data structure: the table. Tables are collections of key-value pairs that are passed by reference and monitored by a garbage collection algorithm. Whereas most modern languages have separate data types for sequences (i.e. arrays or vectors) and records (i.e. tuples or structs), Lua relies on tables with integer- or string-valued keys, and has built-in operations dedicated to these types of tables.

In Lua, when a new variable is created with an assignment statement such as `x = 5`, the underlying effect is to add the key-value pair (`"x"`, `5`) to a special table called the *environment table* (named `_ENV` in Lua 5.2, the version used by DesignDEVS). DesignDEVS modifies environment tables such that newly added keys become state variables of the DEVS model instance. Of course, a variable that is local to a particular event should not be treated as a state variable, and must not persist between events. Fortunately, these local variables are inherently supported by Lua's built-in `local` keyword.

In addition to reinterpreting variable assignments, DesignDEVS expands environment tables by adding the modeling-specific functions in Table 3. The `duration` function creates a time value from a multiplier and a unit (e.g. `duration(8, "minutes")`). With a single string argument, the `duration` function can produce a single unit of the simulation's time precision (`"minimum"`), the maximum representable duration (`"maximum"`), or an infinite duration (`"forever"`). Other key functions include `input`, used in the External Transition tab to access received values; `output` used in the Internal Transition tab to send values; and `elapsed`, used where needed to obtain the time elapsed since the previous event. The `input` and `output` functions rely on the port names specified in the Main tab. Another function worth mentioning is `runscript`, which imports custom Lua scripts from the project folder. This is helpful for managing any atomic

Model	State Initialization	Time Advance	External Transition	Internal Transition
Greeting Generator (2 lines)		`return duration(8, "minutes")`		`output("out", "Hello")`

Counting Generator (4 lines)	`n = 1`	`return duration(8, "minutes")`		`output("out", n)`
				`n = n + 1`

Ideal Processor (18 lines)	`item = nil`
	`inputCount = 0`
	`outputCount = 0`
		`local dt_r =`
		`        duration("forever")`
		`if not (item == nil) then`
		`    dt_r = duration(0)`
		`end`
		`return dt_r`
			`local port, value = input()`
			`if port == "in" then`
			`    item = value`
			`    inputCount = inputCount + 1`
			`else`
			`    error("no port named ’" +`
			`          port + "’")`
			`end`
				`output("out", item)`
				`item = nil`
				`outputCount =`
				`          outputCount + 1`

Table 2. Complete source code for three simple DesignDEVS models. The line counts treat multi-line instructions as one line.

Function	Description
`duration`	Construct a time duration value
`tostring`	Convert a value (incl. table) to a string
`print`	Print arguments on the console
`const`	Make a table permanently read-only
`copy`	Make a deep copy of a table
`runscript`	Load a .lua file from the project folder
`error`	Abort with an error message
`input`	Input a (port, value) pair
`output`	Output a (port, value) pair
`elapsed`	Get time elapsed since previous event
`total_elapsed`	Get time elapsed since simulation start
`remaining`	Get time remaining until planned event
`get_parameter`	Get parameter value
`set_parameter`	Set component parameter value
`get_statistic`	Get component statistic value
`set_statistic`	Set statistic value

Table 3. Modeling-specific functions.

models that do become complex despite the simplicity of the Lua programming language.

5. Coupled model development

According to DEVS theory, coupled models do not expand the set of systems that can be represented. In fact, the closure-under-coupling property assures us that for any DEVS coupled model, a DEVS atomic model can be defined with equivalent behavior[1]. The role of coupling is to provide an abstraction mechanism by which complex models can be defined as hierarchies of simpler models. This eases the development of complex simulations by helping a single user keep his/her code organized, or by helping a team of users collaborate. DesignDEVS emphasizes the potential for coupled models to support collaborative modeling efforts. This influences the types of model hierarchies the tool is intended to support.

Broadly speaking, there are two main approaches for decomposing a system into a hierarchy of models: the *topological* approach and the *functional* approach[24]. In the domain of architectural design, as illustrated at the top of Figure 7, a topological approach decomposes a system into interacting parts such as a building's walls, roof, windows, interior zones, and individual occupants. Zimmermann[25] simulates buildings using strictly this type of system decomposition. The functional approach is illustrated at the bottom of Figure 7. Here a system is decomposed into aspects such as building thermodynamics or occupant behavior, which typically lack well-defined spatial boundaries.

Whereas PowerDEVS[11] and many other simulation environments emphasize a topological approach to system decomposition, DesignDEVS promotes the functional alternative. Topological decompositions are arguably more intuitive, and have an advantage in that they promote greater numbers of simpler atomic models. Yet based on our observations, a domain expert's area of expertise tends to align with one aspect of a system. For example, there are experts in whole-building thermodynamics and experts in occupant behavior, yet it would be rare to find an expert dedicated to modeling all aspects of windows, both from thermal and occupant usage perspectives. We envision scenarios where collaborating experts each model one functional aspect of a building, and the models are later integrated using an encompassing coupled model. Because the component models are developed independently of one another, users must understand and adhere to the principle of delayed binding.

In DesignDEVS, delayed binding is emphasized in the node-link editor for coupled models. DEVS models communicate via messages without explicitly referring to one another, allowing models with similar interfaces to be interchanged. To convey this notion, a node is first drawn, then named, and finally associated with a particular model. The model can be replaced later without deleting the node.

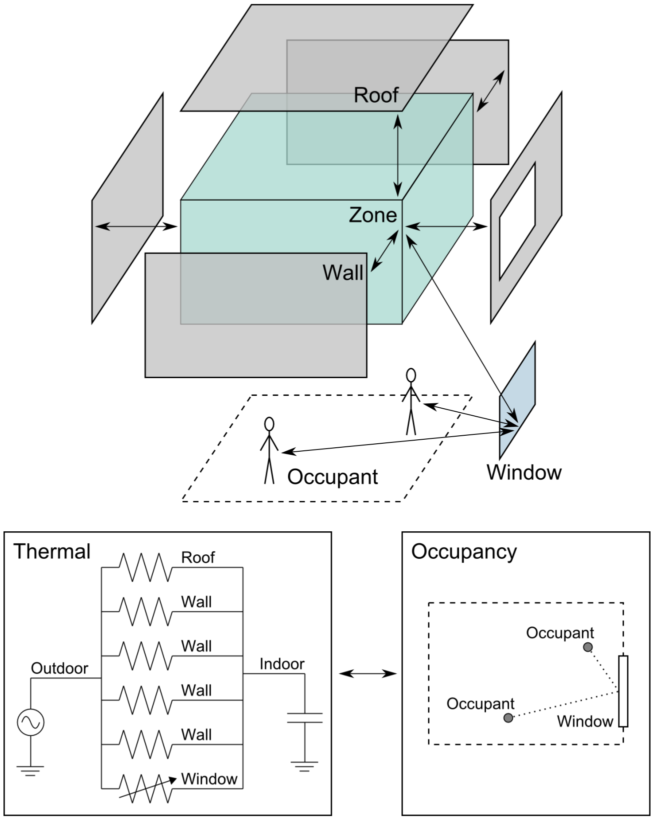


Figure 7. Illustration of models based on topological (top) and functional (bottom) approaches to system decomposition.

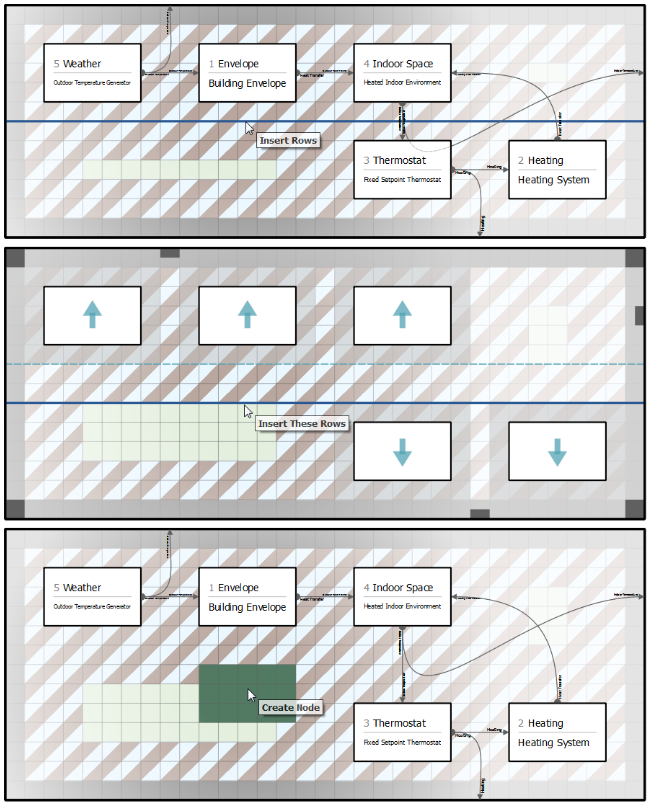


Figure 8. Coupled model expansion and node insertion.

To minimize the risk of important features remaining hidden from users, DesignDEVS places nearly all functionality close to the context of their use instead of in the menu bar at the top. The coupled model editor follows this philosophy to an extreme degree. All node-link editing functions are embedded in the diagram, consistent with the human-computer interaction principle of direct manipulation[26]. The functions are revealed in a tooltip depending on the cursor location. The indicated action is executed simply by clicking; there is no drag-and-drop interaction and no right-click menu.

When the cursor is within the coupled model, regions of the background are filled with diagonal stripes as shown in Figures 8 and 9. Clicking on a blue stripe will expand the model vertically, as in Figure 8 (top), or horizontally, depending on whether the cursor is closer to a horizontal or vertical grid line. Similarly, clicking on a red stripe contracts the model. Some grid cells are solid green, indicating they are sufficiently far from existing boundaries to place a node. Hovering over a green cell shows where a node will be placed upon clicking, as in Figure 8 (bottom).

Once a node is created, red text indicates that it needs a name and model. The user clicks on the red text, types an arbitrary name for the node, and assigns a model to the node by selecting from a pop-up menu. As mentioned in Section 3, a model can only be assigned to a node if the model appears above the coupled model in the list shown at the top left of Figure 3. The user can also re-order nodes by clicking on an integer in the top left corner. Similar to many other Classic DEVS simulators, this ordering of nodes is used in lieu of a full-fledged *Select* function to sequentialize simultaneous events. By default, nodes are ordered according to the sequence in which they were created. Nodes can be deleted or moved by clicking on either the red or blue square that appears at the top-left or bottom-right corner when hovering. It is never possible to produce overlapping nodes or even adjacent nodes that might conceal their surrounding links.

Links indicate how messages flow within a coupled model. To insert a link, one first clicks on a green rectangle on the boundary of a node or the coupled model itself. With the origin of the link determined, green rectangles appear throughout the diagram at every possible link destination. DEVS theory disallows links from a component to itself, so the green rectangles do not appear around the node of origin. Once a link is placed, the ports at either end point can be assigned. Figure 9 illustrates the placement of a link and the final model as a result of the manipulations in both Figures 8 and 9.

Although coupling is the most prominent abstraction technique in a DEVS-based tool, it is not the only form

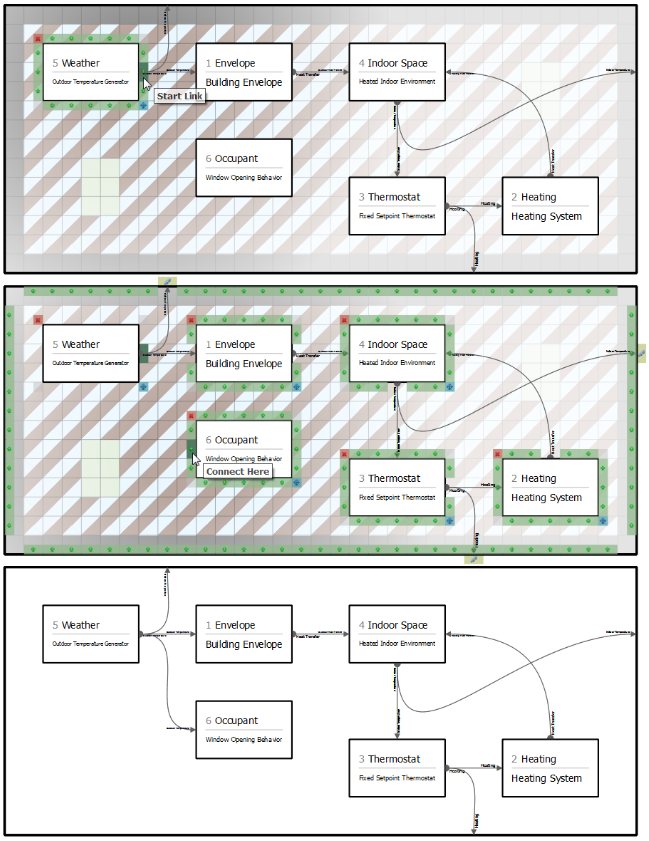


Figure 9. Coupled model link insertion.

of abstraction at the user's disposal. In particular, abstractions provided by the underlying programming language remain useful. With DesignDEVS, the intent is to use functional decompositions and coupling to promote collaboration among domain experts. Users must still be encouraged to separate chunks of procedural code into functions, which can be defined in the Constant Initialization section of an atomic model or in separate files imported using `runscript`.

6. Modeling constraints

The DEVS formalism has a number of theoretical constraints which follow from its mathematical nature. For example, a pre-transition state s is never modified, but rather replaced by a post-transition state s'. DEVS tools differ in how such constraints are enforced. SC-DEVS[27], one of the many non-graphical DEVS-based simulation libraries, is particularly loyal to the theory in that the current `state` (of C++ type `const State&`) is immutable and the new state is a distinct object. If the state requires a considerable amount of memory, the more object-oriented approach of VLE and several other DEVS-based simulators is more computationally efficient. In VLE, state variables can be directly modified by state transition member functions of model-specific C++ classes. Member functions corresponding to the DEVS formalism's time advance and output functions are declared `const` to prevent state changes, consistent with the theory. Similar constraints have been enforced using custom modeling notations[28]. Nevertheless, few DEVS-based simulators fully protect the user from theoretical inconsistencies caused by the use of data references, or *pointers*. DesignDEVS makes use of Lua's extension mechanisms to address the Insidious Pointer Problem and other issues while allowing informative error messages to be delivered to users.

6.1. The Insidious Pointer Problem

As Nutaro[29] writes, an "***insidious problem*** [our emphasis] *with exchanging pointers is that the output object is shared by its producer and all of its recipients.* [...] *In effect, the shared object becomes a hidden channel for communication, and its effects can be unpredictable, and generally undesirable, as the root cause can be difficult to pin down.*" Based on Nutaro's description, we call this the *Insidious Pointer Problem.*

The Insidious Pointer Problem can be regarded as a set of four types of scenarios, each of which contradicts the principle that model instances should interact only through their inputs and outputs. These four basic cases are described below. Components A, B, and **C** are hypothetical model instances in the context of an encompassing coupled model.

1. Component A retains one copy of a pointer, and sends another copy to Component B. Component B then modifies the data referenced by the pointer. This operation in B changes the state of A, even if there is no output sent from B to A.

2. Component A retains one copy of a pointer, and sends another copy to Component B. Component B never modifies the data referenced by the pointer, but merely retains a copy of the pointer. At a later stage, the data referenced by the pointer is modified by A. This operation in A changes the state of B, even if there is no additional output sent from A to B to communicate the change.

3. Component A sends a pointer to Components B and C, without necessarily retaining a copy. Component B then modifies the data referenced by the pointer. This operation in B may change the information that will be received by C from A.

4. Component A sends a pointer to Components B and C, without necessarily retaining a copy. Component B never modifies the data referenced by the pointer, but merely retains a copy of the pointer. At a later stage, the data referenced by the pointer is modified by C. This operation in C changes the

state of B, even if there is never any output sent from C to B.

Problems associated with the exchange of pointers are acknowledged in the testing framework of Li et al.[30], who investigate two simulators and find that both fail at least one test related to Case 4 above.

One solution to the Insidious Pointer Problem is to simply prohibit the exchange of pointers. This risks frustrating a user who wants to output a large numerical array or other memory-intensive data structure while avoiding unnecessary memory allocation and copy operations. A more practical approach is to permit the exchange of pointers while preventing their contents from being modified. In C++, this can be achieved using the `const` type qualifier, though there are three important points to note about this approach. First, the qualifier must be applied to the contents of the pointer (i.e. `const T *`), not the pointer itself (i.e. `T * const`). Second, to address Case 2 (in which the receiving component merely copies the pointer without modifying its contents), the pointer type itself must be non-copyable. Third, the passing of an invalid pointer type should be prohibited by the framework, not the modeler's own code. A promising way to address all three points may be to use template meta programming to restrict all exchanged pointers to a custom pointer type that references only constant data and provides no publicly accessible copy operations. After examining a number of C++ DEVS libraries including Adevs[31], CD++[32], and DEVS++[33], we have not clearly seen a comprehensive solution of this nature, though VLE best handles this issue.

The DesignDEVS approach to pointers, described in Section 6.2 and illustrated in Section 6.3, is suitable for dynamically typed languages, which tend to lack C-like `const` type qualifiers. Our solution strives to allow the contents of exchanged pointers to be modified in cases where it is safe to do so, and delivers a descriptive error message when a pointer assignment or data modification may influence other components.

6.2. Run-time enforcement of constraints

DesignDEVS promotes consistency with DEVS theory in large part through the run-time detection of a broad range of errors, including but not limited to pointer-related errors. Among the simplest errors are those in which a Table 3 function is called in an inappropriate context. For example, if `input()` appears in the Internal Transition tab, the error `"attempt to call 'input' outside of External Transition"` is produced. Clicking the error message in the Console selects the relevant tab and highlights the offending line of code. To detect the error, `input` is implemented as a *closure* (as in functional programming), that captures an instance-specific table, which in turn keeps track of what code is being executed. Other errors are detected through the use of metatables, described below. The entire approach is comprehensive in that no combination of value and pointer assignments should result in undetected side effects that contradict the mathematics of the formalism.

Lua differs from C++ in that instead of declaring variables `const`, one can dynamically control whether a table is constant or mutable. More generally, one can attach or modify *metatables* to customize the language in a number of ways, including but not limited to controlling "constness".

The simplest errors are detected by metatables attached to environment tables. These metatables are aware of what part of a model is being executed. If a state variable is modified in the External Transition or Internal Transition tab, there is no error; but if one attempts to change the variable in the Time Advance tab, a metatable triggers the error `"attempt to reassign a state variable in Time Advance (state changes occur in External and Internal Transitions)"`.

To detect not just a few clearcut errors such as those above, but rather the vast majority of operations that contradict DEVS theory in subtle ways, great attention is paid to the metatables attached to tables defined by the modeler. Recall that in Lua, these user-defined tables are passed by reference and include all sequences and records as well as general collections of key-value pairs. Therefore, careful handling of these tables addresses essentially every data structure and every pointer encountered in a typical DesignDEVS model. The first step is to assign each table one of following types: *raw*, *regular*, *state*, *external regular*, *external state*, *acquired state*, and *constant*. Tables are converted from one type to another depending on the context. Certain operations are prohibited based on the context and the table type.

A *raw table* is a basic Lua table. If a DesignDEVS-specific operation is performed on a raw table, it is converted into a *regular table*. The conversion process triggers an error if the table contains a key value that is another table, a function, or an object that cannot be printed. Importantly, circular references trigger the error `"attempt to record or transmit a table with a circular reference (simulation models require tables that do not reference themselves, not even indirectly)"`.

If a regular table is assigned to a state variable or a table within a state variable, it is converted into a *state table*. If a raw table is assigned in a similar manner, it is silently converted into a regular table before becoming a state table. Altering a state table fails if done in an inappropriate context (e.g. `"attempt to modify a state variable table in Time Advance)"`). This protects not only state variables from undesirable modifications, but also objects referenced by state variables. A related constraint is that a state table may be referenced at most once in a model's state, regardless of how deeply nested it occurs within the model's state variables. A second reference triggers an error (`"attempt to reference a modifiable table in multiple state variables or in multiple places within a single state variable"`), preventing a change within one state variable from affecting another. If a state table contains a state table, and the former is altered such that it no longer contains the latter, the latter is converted back to a regular table.

Three types of tables occur only in input messages received during an External Transition. Their collective purpose is to solve the Insidious Pointer Problem whereby a received data reference—a Lua table, in our case—creates a "*hidden channel for communication*"[29]. When a regular table is received, it becomes an *external regular table* which, to avoid influencing other recipients, cannot be altered (`"attempt to modify a table just received from another model instance"`). This addresses Case 1 and Case 3 of Section 6.1. However, an external regular table can be stored in the recipient's state. The table then becomes an *acquired state table*, and can be modified in a future External Transition or Internal Transition. If a state table or acquired state table is received in a message, it is an *external state table*. To address Case 1, such tables cannot be altered (`"attempt to modify a state variable owned by another model instance"`). To address Case 2, they cannot be stored (`"attempt to reference a modifiable table in state variables of multiple model instances"`).

Importantly, when an external regular table is acquired by one component and becomes an acquired state table, it is regarded by all subsequent receiving components as an external state table. This means that a table can only be acquired by at most one receiver, addressing Case 4. The policy relies on the fact DesignDEVS processes all events in sequence. The external events associated with a single message are executed according to the user-defined ordering of components—the same ordering used to resolve simultaneous internal events. A more general approach would be to use a full-fledged *Select* function to order both types of events.

State table acquisition complies with DEVS theory in the sense that, if it occurs error-free in a DesignDEVS model, the components influence one another only via messages and the message values are unaffected. Although one receiving component can influence another by converting the received table into a state table, this interaction will either (a) have no effect, or (b) invalidate the model by producing an error. Nevertheless, transferring the ownership of a table should be regarded as an advanced optimization technique, and used sparingly.

Any table can be converted to a *constant table* using the `const` function. Once made constant, a table can never be altered (though it can be copied, and the copy can be modified). Multiple references to constant tables can exist both within a single model instance and among communicating instances.

If desired, the user can avoid memory sharing by copying a table using the `copy` function. Some DEVS tools address the Insidious Pointer Problem by performing such deep copies by default. For sake of computational efficiency, DesignDEVS accommodates a variety of communication patterns involving pointers, yet still circumvents their problematic effects. The solution is unique, and provides a means to inform users about the constraints being enforced.

6.3. *Illustration of run-time enforcement*

Figure 10 illustrates how the various types of tables collectively address the Insidious Pointer Problem. The diagram provides a simplified representation of the two hypothetical atomic models assigned to Components A and B. The lines of code shown are not necessarily consecutive and would not necessarily appear in the same section of either model. The image is merely intended to show the types of operations that trigger table conversions or table-related errors.

Let us start at the top of Figure 10, in Component A. Line 1 is a basic Lua instruction that creates a raw table and stores it in the variable `X`. Because DesignDEVS overrides Lua's `print` function, Line 2 converts the raw table into a regular table. Here an error would be generated if the table does not meet certain requirements, such as the absence of circular references. Line 3 adds the regular table to the state variable `state_var_1`. This causes it to be converted into a state table.

To understand how Cases 1 and 2 of the Insidious Pointer Problem are handled, consider the first output in Figure 10. It is received by Component B on Line 4. Upon receipt, the state table is treated as an external state table. As illustrated on Line 5, an attempt to modify an external state table triggers an error. Otherwise Component B would alter the state of Component A in contradiction with DEVS theory (see Case 1 of Section 6.1). Also, as seen on Line 6, Component B cannot store the external state table as doing so would allow its own state to be modified by Component A without any subsequent message (Case 2).

To observe the handling of Cases 3 and 4 of the Insidious Pointer Problem, we shift our attention back to Component A. On Line 7 the state table is removed from the state, and is therefore converted back into a regular table. Let us assume that the variable `X` exists and maintains a reference to this table. When the regular table is received by Component B on Line 8, it becomes an external regular table. As indicated on Line 9, an external regular table may not be modified as such a change could influence other recipients of the same output (Case 3). For example, if the same output were subsequently re-

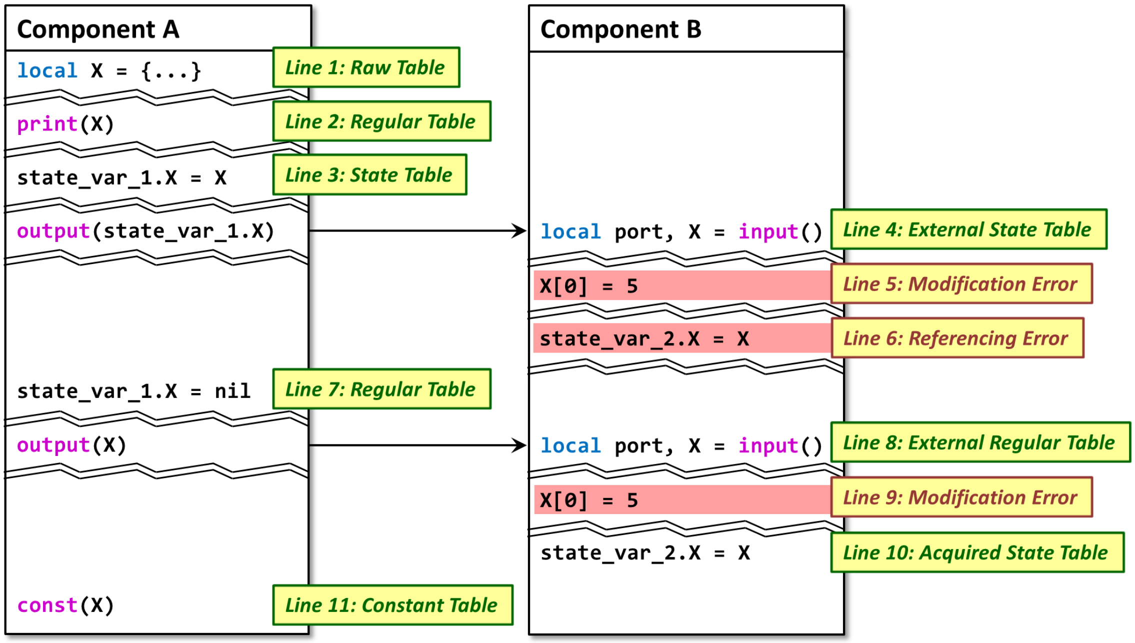


Figure 10. An illustration of various atomic model operations, the types of tables they produce, and how the type of table restricts further operations as a means of addressing the Insidious Pointer Problem.

ceived by a Component C (not shown), then Line 9 in B would threaten to influence C. Although an external regular table may not be modified, it can be stored as shown on Line 10. In that case it is converted into an acquired state table. An acquired state table also cannot be modified; however, at the end of the External Transition it will be converted into a state table, allowing for subsequent modifications. Because only one component can acquire a table, these future modifications will not affect other components (Case 4). For example, if the same output were subsequently received by a Component C, then C would regard the table as an external state table that can be neither modified nor stored.

Finally, at the bottom of Figure 10, Line 11 explicitly converts the table into a constant table. The table is thereafter permitted to be referenced in any context, but never modified. A constant table can be exchanged freely among components, and stored in multiple state variables, without triggering errors.

7. Best practices

In any theory-based simulation tool, practical considerations such as user convenience and computational efficiency sometimes conflict with the underlying formalism[34]. Accordingly, we find that some DEVS principles are appropriate to enforce through constraints such as those in Section 6, whereas other principles can be encouraged by defining and communicating a set of best practices. An analogy can be made with object-oriented programming. It is widely acknowledged that an object's data should be encapsulated by declaring member variables private[35]. Yet language designers do not enforce this rule. In most popular object-oriented languages, member variables may be declared public, though software engineers frequently declare them private in compliance with a well-established best practice. Similar types of best practices must also be considered in a modeling and simulation context as an alternative to strict enforcement.

7.1. Output function

According to DEVS theory, the output function λ and the internal transition function δ_{int} are invoked in sequence; their effects are considered to be distinct, but they occur at the same point in simulated time. This invocation sequence can be adhered to in DesignDEVS, but only at the user's discretion. DesignDEVS absorbs the output function into the Internal Transition, and accommodates the production of more than one output in a single state transition.

The motivation for merging the implementations of λ and δ_{int} is that the two functions often involve same intermediate calculations. If the functions are kept separate, as they are in most existing DEVS-based tools, these calculations are likely to be performed twice instead of once. Alternatively, the software can allow the output function to make state changes, but this also contradicts DEVS theory and may confuse users who realize that they can move code between the two functions without changing simulation results.

It is important to note that the theoretical behavior of a Classic DEVS model is unaffected by the co-evaluation of λ and δ_{int} (in contrast with multiple outputs, which does introduce a behavioral discrepancy with the theory). There is no possibility of an intervening event that (a) is processed between λ and δ_{int}, and (b) has the potential to causally affect δ_{int}. This observation is partly due to the rule that no component can send a message directly to itself, and partly a result of the fact the theory does not account for casual relationships introduced by technological side effects.

We have observed that one important class of model is particularly troublesome to implement with separated λ and δ_{int} functions: numerical integrators in which outputs coincide with integration steps. In such models, it is necessary to perform an integration step in order to output the next value; the same value must then be stored in the model's state to be used in future integration steps. The simplest example is the classic Ramp model of *Theory of Modeling and Simulation*[1], chapter 4, which integrates a piecewise constant function. The integration step ($position + \sigma \cdot input$) appears first in λ to supply the output value, and again in δ_{int} to be stored. This suggests an implementation in which the calculation is unnecessarily repeated. Complex numerical integrators involving thousands of equations can be found in a multitude of scientific disciplines, and are among the most computationally demanding simulation models. The experts who develop such models are unlikely to appreciate the theoretical benefits of a separate λ function, but they will notice the practical costs. To achieve widespread adoption, DEVS must be regarded as an efficient implementation technique for nearly any time-based simulation method.

Whereas the absorption of λ is motivated by efficiency concerns, the reason DesignDEVS permits multiple outputs is simply a matter of user convenience. If it is necessary to output one value on one port, followed immediately by another value on a different port, Classic DEVS requires the model to be designed such that two events are guaranteed to occur back-to-back. This requires a certain amount of technical code to be added to multiple atomic model functions. It is much easier to write `output(...)` multiple times in the Internal Transition, and DesignDEVS provides this option. The outputs are transmitted such that each is received by a separate External Transition event, and the order of the outputs determines the order of the receiving events.

Care must be taken not to frustrate domain experts to the point that they abandon formalism-based tools and revert to ad-hoc simulation development approaches. At the same time, such tools must promote theoretical principles to enough of an extent that they are worth adopting in the first place. Furthermore, the tools must be suitable for formal methods experts and students, users who may want to respect and/or learn theory-based conventions. We therefore propose a best practice intended for experienced and aspiring DEVS experts.

1. Formulate the atomic model specification according to Classic DEVS conventions. In mathematical form, the separation of λ and δ_{int} is not troubling from an efficiency standpoint. The use of a distinct λ simplifies mathematical analyses such as a proof that an atomic model is formally legitimate. Legitimacy means that in the absence of inputs, simulated time necessarily advances towards ∞ without stopping or converging[1]. Note that the formula for legitimacy, shown below, depends on δ_{int} but not λ.

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

2. Still adhering to Classic DEVS, transform the specification such that operations that would be computationally expensive are only invoked once. This may require that a succession of two or more internal transitions are triggered at the same point in simulated time. Ideally one would prove that the original and optimized specifications are equivalent.

3. In DesignDEVS, draw a horizontal line through the Internal Transition. In Lua, the token `--` initiates a comment. It follows that a horizontal line of hyphens (`--------------`) is itself a comment, and can be inserted on any line.

4. Translate the λ of the specification in Step 2 into Lua code, and place the code in the Internal Transition *above* the horizontal line. The resulting code should contain no state changes, and the `output` function should be invoked either just once or not at all. If there is an invocation of `output`, it should occur at the end, just above the horizontal line. These self-imposed restrictions honor the fact that according to DEVS theory, λ can result in either a single output or $\varnothing$ (no output).

5. Translate the δ_{int} of the specification in Step 2 into Lua code, and place the code in the Internal Transition *below* the horizontal line. Here state changes are permitted, but the `output` function is never called. Any local variable declared in the λ section above the horizontal line should not be referenced in the δ_{int} section below the horizontal line.

Note that the horizontal line in Step 3 helps the advanced user voluntarily separate λ and δ_{int}. Had DesignDEVS enforced this convention, the code above and below the horizontal line would instead appear in separate tabs.

Model	State Initialization	Time Advance	External Transition	Internal Transition

Convenient
(14 lines)
	n = 0
	x = 50
	y = 10
	delay = time_step
		return delay	local port, value = input()
			x = value
			delay = delay - elapsed()
				y = call_me_once_per_count(n, x, y)
				output("B", y)
				output("C", x)
				output("D", n)
				n = n + 1
				delay = time_step

Conventional
(44 lines)
	n = 0
	x = 50
	y = 10
	delay = time_step
	phase = 1
	deferred_x = nil
		return delay	local port, value = input()
			if phase == 1 then
			    x = value
			else
			    deferred_x = value
			end
			delay = delay - elapsed()
				local port, value = nil, nil
				if phase == 2 then
				    port, value = "B", y
				elseif phase == 3 then
				    port, value = "C", x
				elseif phase == 4 then
				    port, value = "D", n
				end
				if phase > 1 then
				    output(port, value)
				end
				-- 	
				if phase == 1 then
				    y = call_me_once_per_count(n, x, y)
				    delay = duration(0, "seconds")
				    phase = 2
				elseif phase == 2 then
				    delay = duration(0, "seconds")
				    phase = 3
				elseif phase == 3 then
				    delay = duration(0, "seconds")
				    phase = 4
				elseif phase == 4 then
				    n = n + 1
				    delay = time_step
				    phase = 1
				    if not deferred_x == nil then
				        x = deferred_x
				        deferred_x = nil
				    end
				end

Table 4. A Convenient model that takes advantage of DesignDEVS features is shown alongside an equivalent Conventional model that adheres to best practices according to DEVS theory.

The example in Table 4 illustrates both the separation of λ and δ_{int} as well as the reason why DesignDEVS does not strictly enforce this convention. Observe in the Convenient model a state variable `n` which counts internal transitions, and a computationally intensive function `call_me_once_per_count` that should never be invoked twice with the same arguments. At every time step, the model evaluates this function, then outputs its result `y`, followed by the value of the previous input `x`, followed by `n`. By permitting repeated invocations of `output` within the Internal Transition, the model can be implemented in 14 reasonably intuitive lines of code.

Equivalent to Convenient in terms of how a sequence of (simulated time, value) input pairs is translated into outputs, the Conventional model adheres to best practices by separating the implementations of λ and δ_{int}. The Internal Transition features a horizontal line, above which there are no state changes and below which there are no calls to `output`. To achieve the desired behavior, the Internal Transition must be executed four times in quick succession with different operations on each pass. A state variable named `phase` is added to ensure the operations unfold in the proper order. When `phase = 1`, the `output` function is skipped such that `call_me_once_per_count` can be called first. The next three passes (`phase = 2`, `phase = 3`, `phase = 4`) each call `output` with the appropriate port and value combination. At the end of the `phase = 4` pass, the state variables are updated.

Another complication with the Conventional model is the possibility the `output("C", x)` call is preceded by an input. To prevent the input from altering behavior, it is temporarily stored in a variable called `deferred_x`, and only assigned to `x` at the end of the `phase = 4` pass through Internal Transition. By directly supporting multiple instantaneous outputs, as shown in the Convenient model, DesignDEVS shields domain experts from the inconvenience of deferring inputs in this manner. As a trade-off, these users may be somewhat less likely to acquire a deep understanding of the formalism unless they study the literature and adopt best practices.

It should be observed that the example in Table 4 depicts a worst-case scenario for Classic DEVS conventions. The best practice of separating λ and δ_{int} will not typically result in a 3-fold increase in code. Nevertheless, DEVS-based tool developers must remain aware that domain experts will appreciate only the practical benefits of applying the theory. If the formalism's positive attributes appear overwhelmed by its perceived inconve-

Model	Components	Initialization

Bank	1. **Machine Lineup**
	(Model: Customer Lineup)
	2. **Machine**
	(Model: Bank Machine)
	3. **Teller Lineup**
	(Model: Customer Lineup)
	4. **Teller**
	(Model: Bank Teller)
		`exponential = runscript("statistics.lua").exponential`
		
		`local capacity_ml = get_parameter("Machine Lineup Capacity")`
		`set_parameter("Machine Lineup", "Capacity", capacity_ml)`
		
		`local dt_average_m = get_parameter("Machine Average Service Duration")`
		`set_parameter("Machine", "Average Service Duration", dt_average_m)`
		
		`local rate_ex = get_parameter("Exchange Rate")`
		`set_parameter("Machine", "Exchange Rate", rate_ex)`
		`set_parameter("Teller", "Exchange Rate", rate_ex)`
		
		`local naccts = get_parameter("Number of Accounts")`
		`local average = get_parameter("Average Balance")`
		`balances0 = {}`
		`for account=1,naccts do`
		`    balances0[account] = math.floor(exponential(average) + 0.5)`
		`end`
		`const(balances0)`
		`set_parameter("Machine", "Initial Balances", balances0)`
		`set_parameter("Teller", "Initial Balances", balances0)`
		
		`local capacity_tl = get_parameter("Teller Lineup Capacity")`
		`set_parameter("Teller Lineup", "Capacity", capacity_tl)`
		
		`local dt_average_t = get_parameter("Teller Average Service Duration")`
		`set_parameter("Teller", "Average Service Duration", dt_average_t)`

Table 5. The components and initialization code for a DesignDEVS coupled model representing a bank machine, a bank teller, and two customer lineups at a bank.

niences, as they do in the Table 4 example, practitioners may turn away from theory-based approaches and revert to the ad-hoc modeling practices that continue to dominate most scientific and engineering domains. The issues emphasized in this example do arise on occasion, in our experience.

Simulation environments based on Parallel DEVS must take a different approach, since in this formalism an invocation of λ does not always precede δ_{int}. Yet the problem of intermediate calculations is still relevant, and deserves attention. Part of the motivation for Parallel DEVS is to achieve greater parallelism than is possible under Classic DEVS. This practical benefit is put at risk if a computationally expensive function must be invoked to both produce an output and complete the subsequent state change. Both Classic and Parallel DEVS environments require best practices to guide users on how such situations should be addressed.

7.2. Parameters and statistics

Given the popularity of object-oriented programming, domain experts may understand encapsulation even if they are unfamiliar with other DEVS-related principles. The DEVS formalism's inputs and outputs encapsulate the state of an atomic model during a simulation. Most DEVS-based tools incorporate parameters, which encapsulate state at the beginning of a simulation. DesignDEVS also includes statistics, which provide information at the end of a simulation without exposing the state.

Unfortunately, the inclusion of parameters and statistics may lead users to place configuration and analysis code in their models, contradicting the theoretical principle that experiment-related elements be confined to an *experimental frame*[1]. In DesignDEVS, the principle of encapsulation is given priority and enforced through modeling constraints, whereas the separation of model and experiment relies on the users' adherence to best practices. We begin this discussion by describing how parameters and statistics are treated in DesignDEVS.

Many simulation environments allow users to directly modify the parameters of the components of a coupled model. Unfortunately, this violates the principle of encapsulation by exposing the composition of a coupled model in an external context. DesignDEVS takes a different approach, enforcing encapsulation by giving every coupled model an Initialization tab. The modeler may read the parameters of the coupled model (using `get_parameter`) and derive the parameters of the components (using `set_parameter`). With this mechanism, it is possible to centralize pre-simulation processing operations, as well as ensure that two components receive the same parameter value where needed.

Table 5 gives a simple example of how parameters are handled in DesignDEVS coupled models. Here Bank is a coupled model consisting of four components: Machine Lineup, Machine, Teller Lineup, and Teller. As shown on the right of Table 5, the parameters of the four components are automatically derived from the parameters of the encompassing Bank model through its Initialization code. In many cases, a coupled model parameter is simply redirected to one of the components. For example, the Bank model's `"Machine Lineup Capacity"` parameter becomes the Machine Lineup component's `"Capacity"` parameter. Similarly, the Bank model's "Teller Lineup Capacity" parameter becomes the Teller Lineup component's "Capacity" parameter. Yet not all parameters have a one-to-one mapping. The Bank model's "Exchange Rate" parameter is assigned to both the Machine and Teller components. Observe that the Bank model's "Average Balance" parameter is used to randomly generate initial account balances, which are then sent to both the Machine and Teller.

In a DesignDEVS atomic model, parameters are available only in the Constant Initialization tab. A parameter can be assigned to a constant, if desired, or otherwise constants can be computed from the parameter values. These constants are actually mutable within the tab, but are automatically fixed prior to State Initialization.

In the same way that parameters encapsulate the state and composition of models at the beginning of a simulation, statistics support encapsulation at the end. In a DesignDEVS atomic model, statistics are assigned in the Finalization tab, which mirrors a similar element in PowerDEVS. As with a C++ destructor, Finalization code may be used to release access to resources, though its primary purpose in DesignDEVS is to support statistics. Not all users require statistics, but those that do benefit from a built-in reporting mechanism. Without statistics, one concern is that some users will resort to a hack in which state information is written to a file and loaded by a separate process, breaking encapsulation. The risk is increased due to the difficulty of capturing information about a model's final state, including the final elapsed duration, in a regular output message. Similar to parameters, the statistics of a coupled model's components cannot be accessed directly; instead, they are used to derive the coupled model's own statistic values.

Although DesignDEVS incorporates parameters and statistics in a manner that completely encapsulates the state of an atomic model and the composition of a coupled model, these conventions do make it convenient for users to mix experiment- and model-related code in contradiction with the tenets of the formalism. The treatment of statistics is of particular concern, as it is hard to distinguish between an experiment-independent statistic and one specific to a particular application of the model. This leads us to suggest the following best practice.

1. Users should distinguish between *system models*, *configuration models*, *analysis models*, and *experiment models.*

2. A *system model* is the most standard type of model: one that represents a system. It should have no statistics and no experiment-specific elements.

3. A *configuration model* produces input data specific to an experiment and a system, but independent of how the system is modeled. It is analogous to a *generator model* in classic experimental frame examples[1]. It should have no statistics.

4. An *analysis model* processes output data in a manner that is specific to an experiment and a system, but independent of how the system is modeled. It is analogous to a *transducer model* in classic experimental frame examples[1]. It may have statistics.

5. An *experiment model* is a coupled model that contains a configuration, system, and analysis model connected in sequence. It represents a simulation-based experiment, and may have statistics.

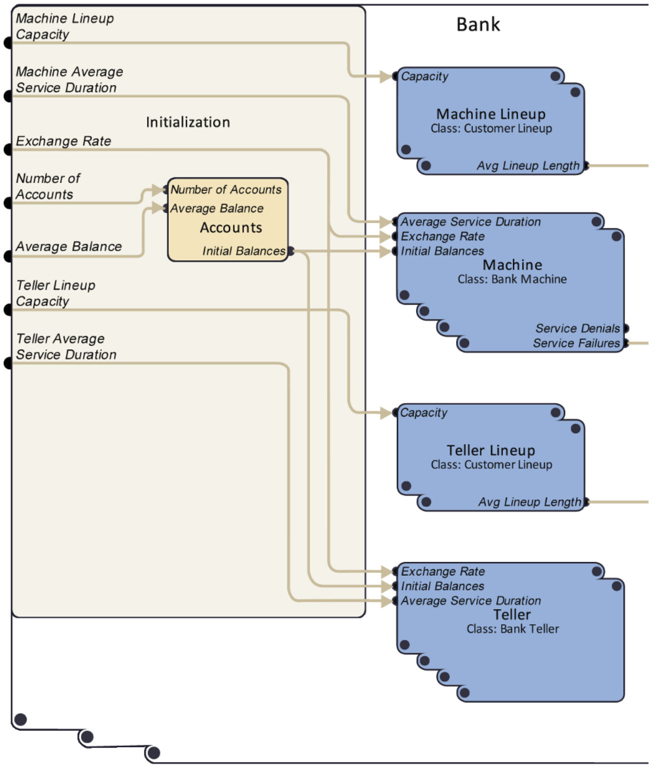


Figure 11. A dataflow representation of the DesignDEVS initialization function of Table 5, as illustrated by Maleki et al.[36].

If best practices are followed, the DesignDEVS approach to parameters and statistics should lead to both well-organized experimentation code and fully encapsulated model compositions. However, experience applying DesignDEVS reveals that coupled model initialization code, such as that in Table 5, can be tedious to write. This was part of the motivation for the work of Maleki et al.[36], who explore data flow programming[37] in the pre- and post-simulation phases of a digital experiment. Figure 11 illustrates what a user interface might look like with data flow elements incorporated into a coupled DEVS model. This work represents a new direction for DEVS-based simulation environments that prioritize the principle of encapsulation as well as modeler convenience. The benefits of data flow programming are also discussed by Doore et al.[38] in the context of modeling and simulation education.

7.3. Time resolution

In DesignDEVS, every model and every simulation has a Time Resolution field which defaults to N/A, but can be changed by the user to 10^6 years, 10^3 years, years, days, hours, minutes, seconds, 10^{-3} seconds, 10^{-6} seconds, [...], 10^{-36} seconds. The Time Resolution of a coupled model cannot be coarser than that of any of its components. The Time Resolution of a simulation cannot be coarser than that of its associated model.

A simulation's Time Resolution acts as a precision level to which all time durations are rounded, even if the submodel that produced the duration has a coarser Time Resolution. The modeler can control the rounding method by supplying an optional third argument to the `duration` function: `"floor"`, `"ceil"`, `"halfup"` (default), `"halfdown"`, `"halfeven"`, or `"halfodd"`. This rounding of time values is not to be confused with time discretization. Discrete-time simulation implies a common time step separating consecutive events. Rounding is simply an unavoidable consequence of the fact real-valued durations cannot all be represented on a computer.

OMNeT++[39] and ns-3[40] are similar to DesignDEVS in that they use a fixed-point representation of simulated time, and the time precision is associated with the simulation. The difference is that DesignDEVS allows each model to impose a bound on the precision level, preventing time-related rounded errors from surpassing a certain degree of severity. If a model is included in a simulation, the simulation's time precision can be finer than that of the model, but no coarser.

There are a number of reasons why DesignDEVS does not dispense with explicit time resolution/precision levels by incorporating a floating-point representation of simulated time. Floating-point time values produce rounding errors in the simulator, where they must be added and subtracted. This unnecessarily denies users the option of performing error-free discrete-event simulations in applications where errors can be avoided[41]. Vicino et al.[42] list many other problems with floating-point time representations. OMNeT++ was originally developed using floating-point time, yet switched to a fixed-point representation for similar reasons as explained by Varga[39]: "*Why did OMNeT++ switch to int64-based simulation time? double's mantissa [the coefficient, excluding the sign and hidden bits] is only 52 bits long, and this caused problems in long simulations that relied on fine-grained timing, for example MAC [media access control] protocols. Other problems were the accumulation of rounding errors, and non-associativity (often $(x+y)+z \neq x+(y+z)$, see Goldberg[43]) which meant that two double [64-bit floating-point number] simulation times could not be reliably compared for equality.*".

The idea of associating a model with an explicit measure of time granularity was recommended more than 20 years ago[44]. Yet attention must be given to how DesignDEVS users should choose a model's Time Resolution. The best practice is to start by applying a theory, and the most relevant theoretical method was developed by Goldstein et al.[41] in conjunction with the DesignDEVS software. The method involves the derivation of a model's *optimal time quantum* from its formal specification, a procedure that may require induction proofs and other mathematical techniques that are likely too unfamiliar and too time-consuming for most domain experts. Fortunately, a simple rule takes the place of such analyses for models with the following property.

$$ta(s_i) \in \{\Delta t_0, \Delta t_1, \ldots, \Delta t_m, 0, \infty, ta(s_{i-1}) - \hat{e}\} \quad (1)$$

In (1), s_{i-1} is a state ($s_{i-1} \in S$), s_i is a state that can possibly succeed s_{i-1} (either $s_i = \delta_{int}(s_{i-1})$ or $s_i = \delta_{ext}(s_{i-1}, e, x)$ for some e and x where $0 \le e \le ta(s_{i-1})$ and $x \in X$), and the variables $\Delta t_0, \Delta t_1, \ldots, \Delta t_m$ are positive rationale numbers representing duration-valued constants known at the outset of a simulation run. The variable $\hat{e}$ is the duration of simulated time elapsed in state s_{i-1} before the model transitions to s_i ($\hat{e} = e$ for external transitions; otherwise $\hat{e} = ta(s_{i-1})$). The simple rule mentioned above is that, if a model satisfies (1), its optimal time quantum is the greatest common divisor (GCD) of $\Delta t_0, \Delta t_1, \ldots, \Delta t_m$. Hence a discrete-time simulation model obeying $ta(s_i) = \Delta t_s$ has an optimal time quantum equal to its time step Δt_s. A classic processor model, as defined by Zeigler et al.[1], satisfies $ta(s_i) \in \{\Delta t_r, \infty, ta(s_{i-1}) - \hat{e}\}$ where Δt_r is a fixed response duration separating an input from its processed output. Since this property adheres to (1), the optimal time quantum is Δt_r. Where feasible, the Time Resolution in DesignDEVS should be the coarsest option that evenly divides the optimal time quantum.

Only the optimal time quantum at the topmost level of a closed system is guaranteed to evenly divide all elapsed durations $\hat{e}$ and time advance durations $ta(s_i)$. Models similar to the classic processor have finite positive optimal time quanta that, due to the unknown timing of inputs from external sources, do not necessarily divide the real-valued durations they experience. The optimal time quanta for such models are nevertheless useful for calculating the optimal time quantum of an encompassing coupled model, which can then be used for calculations at the next level up, and so on. The optimal time quantum of a DEVS coupled model is simply the GCD of that of its component models. Accordingly, the Time Resolution of a DesignDEVS coupled model should be the finest Time Resolution of any submodel, which is also the GCD due to the restricted set of allowable resolution levels. The finest Time Resolution will then propagate upward toward the simulation level, which is essentially a closed system on top of the model hierarchy.

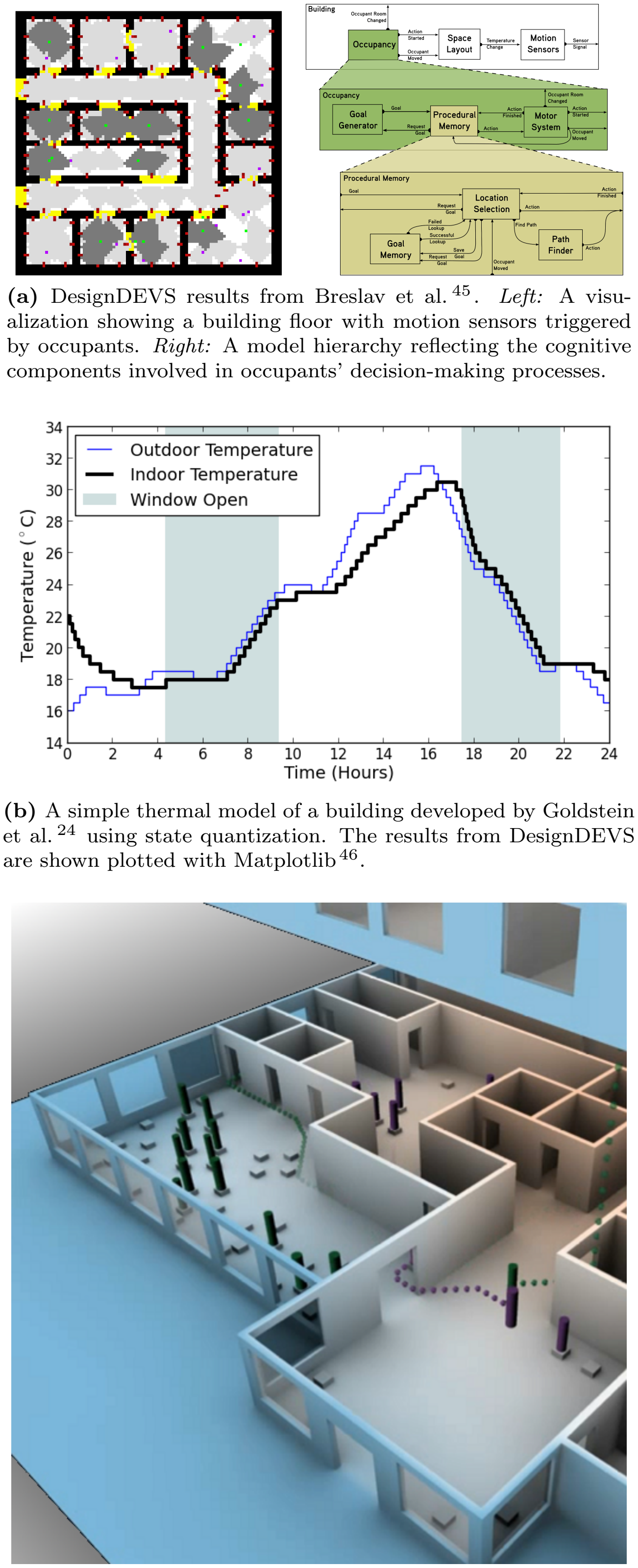
Some models have an optimal time quantum of infinity. In the vast majority of cases, these are models for which the Time Advance function would return either `duration(0)` or `duration("forever")`. The best practice is to assign these models a Time Resolution value of N/A.

Whereas models such as the classic processor experience non-quantized durations only due to external influences (assuming parameters such as Δt_r are rational), certain models do so of their own accord. Typical examples include generators that produce outputs at randomly sampled intervals, and quantized integrators. For such a model, which has an optimal time quantum of zero, the best practice is to select a Time Resolution well below its time scale. It is reasonable for instance to approximate the time scale based on intuition, divide by one million, and round down to the next allowable option. Consider a model that produces outputs at intervals randomly generated from an exponential distribution with a 20-nanosecond mean duration. The optimal time quantum is zero, but the time scale can be regarded as 20 ns. Dividing by one million and rounding down, the Time Resolution of the model would be set at femtoseconds. It is worth noting that in this case, the longest representable duration is exactly 9.007199254740991 seconds, the value given by `duration("maximum")`. Choosing an excessively fine precision constrains the length of a simulation run.

An alternative to the best practice is to select a fine Time Resolution for any model, regardless of its optimal time quantum. In that case a discrete-time model with a time step of 5 seconds might be given a microsecond Time Resolution. If this were the best practice, then the label "precision" would be more appropriate than "resolution" (see Goldstein et al.[41] for definitions of these terms). The fact it is called Time Resolution may remind users that they are free to supply this field with a discrete-time model's uniform time step, or the largest factor of that time step that is available as an option.

8. Applications

Although the software has no discipline-specific features and can be applied to any domain, the name "DesignDEVS" reflects an early focus on building design projects. Similar to other application areas, modern architectural design is characterized by a diverse set of mostly independent simulation-based analysis tools. Theory-based approaches have the potential to support the development of an array of next-generation predictive models that can be readily integrated in various combinations. Here we list several simulation efforts supported by DesignDEVS, all of which have a relationship with architecture.



(a) DesignDEVS results from Breslav et al.[45]. *Left:* A visualization showing a building floor with motion sensors triggered by occupants. *Right:* A model hierarchy reflecting the cognitive components involved in occupants' decision-making processes.

(b) A simple thermal model of a building developed by Goldstein et al.[24] using state quantization. The results from DesignDEVS are shown plotted with Matplotlib[46].

(c) A virtual hotel simulated by Goldstein et al.[47] using DesignDEVS and visualized with Maya[48]. The indoor temperature distribution (surface coloring) influences the window-opening behavior of occupants (cylinders), which in turn affects indoor temperature.

Figure 12. Applications of DesignDEVS.

Breslav et al.[45] use DesignDEVS to simulate the efficacy of an occupant sensor network as a function of the density of motion detectors in a building environment. This project combines a 2D cellspace model of a building's floor plan and occupant locations (Figure 12a, left) with a hierarchical human behavior model inspired by human cognition research (Figure 12a, right).

Goldstein et al.[24] present a number of simple DesignDEVS models illustrating different approaches for solving heat transfer equations in the context of buildings with active and passive heating/cooling. One approach, referred to as a speculative strategy, involves models which communicate future predictions multiple times per time step before converging on a set of mutually consistent values. This allows longer time steps. The quantized state approach is also demonstrated for the coupled thermodynamics and occupancy model depicted in Figure 7 of Section 5, the results of which are shown in Figure 12b.

Gunay et al.[49] apply DesignDEVS to explore the effect of time steps on stochastic models involving occupants, offices, and temperature control. The results demonstrate the advantages of discrete-event simulation over the conventional discrete-time approach.

The most complex DesignDEVS model to date is a discrete-space, discrete-event hotel simulation developed by Goldstein et al.[47]. A 3D animation of the results is shown in Figure 12c. The merging of λ and δ_{int} is exploited by a submodel representing heat diffusion. A fine-resolution array of temperatures is computed once, stored in the submodel's state, and communicated to another submodel responsible for occupant comfort. The initialization function of the overall coupled model centralizes the loading of datasets, which are then distributed in memory to multiple submodels. These practical features helped the modelers remain focused on domain-specific modeling tasks.

9. Conclusion

The DesignDEVS simulation environment contributes to the ongoing exploration of how to best incorporate modeling and simulation theory into practical tools. Its focus is on discrete-event simulation, and in particular the DEVS formalism which generalizes numerous other modeling formalisms. The guidelines that emerge from a detailed look at DesignDEVS are worth considering regardless of the tool developers' paradigm of interest. The first guideline is to express elements of the theory, such as the separation of model and simulator, in the design of the user interface. The second guideline emphasizes the minimization of unnecessary boilerplate code. Third, the model coupling interface should be based on a vision for how this key abstraction mechanism is intended to be applied. Fourth, modeling constraints should be used where appropriate to enforce and communicate aspects of the theory. The fifth guideline is to recommend a set of best practices for theoretical principles that are either impossible or impractical to strictly enforce.

A decision on whether to address a theoretical principle using strict enforcement or using a best practice requires thorough and deliberate consideration. Although it is tempting to base such a decision on the ease with which the principle of interest can be enforced, we strongly recommend that simulation tool developers consider the tradeoffs from the user's perspective. It would not have been difficult to separate output values from state transitions, as indicated by the theory. Yet we chose to leave this separation as a best practice for users who appreciate the theoretical benefits of this convention. By contrast, the Insidious Pointer Problem was very difficult to circumvent, yet we incorporated a solution for both theoretical and practical reasons. Developers of future DEVS-based simulation environments may well choose to keep the output function separate from the internal transition. In that case, our advice is to establish best practices on how to avoid redundant calculations, especially for numerical integrators in which integration steps and outputs coincide. Future tools may also permit the unrestrained sharing of pointers among models. In that case, best practices are needed to reduce the likelihood of producing invalid results due to the accidental sharing of memory. Attention to the practical aspects of formalism-based simulation environments will help increase the utilization of scalable modeling and simulation practices in the disciplines that need them the most.

References

1. Zeigler BP, Praehofer H, and Kim TG. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, San Diego, CA, USA, second edition, 2000.

2. Goldstein R, Breslav S, and Khan A. DesignDEVS: Reinforcing theoretical principles in a practical and lightweight simulation environment. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2016.

3. Ierusalimschy R, De Figueiredo LH, and Celes W. Passing a language through the eye of a needle. *Communications of the ACM*, 54(7):38–43, 2011.

4. Fritzson P and Bunus P. Modelica – a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings of the Annual Simulation Symposium (ANSS)*, 2002.

5. MathWorks. *Simulink: Dynamic System Simulation for MATLAB*. 2000.

6. Ptolemaeus C, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

7. Vangheluwe HLM. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design (CACSD)*, 2000.

8. Goldstein R. DEVS-based dynamic simulation of deformable biological structures. Master's thesis, Carleton University, 2009.

9. Van Schyndel M, Wainer GA, Goldstein R, Mogk J, and Khan A. On the definition of a computational fluid dynamic solver using cellular discrete-event simulation. *Journal of Computational Science*, 5(6):882–890, 2014.

10. Van Mierlo S, Van Tendeloo Y, Mustafiz S, Barroca B, and Vangheluwe H. Explicit modelling of a Parallel DEVS experimentation environment. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, Alexandria, VA, USA, 2015.

11. Bergero F and Kofman E. PowerDEVS: A tool for hybrid system modeling and real-time simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 87(1-2):113–132, 2011.

12. Zengin A and Sarjoughian H. DEVS-Suite simulator: A tool teaching network protocols. In *Proceedings of the Winter Simulation Conference (WSC)*, 2010.

13. Sarjoughian HS and Elamvazhuthi V. CoSMoS: A visual environment for component-based modeling, experimental design, and simulation. In *Proceedings of the International Simulation Tools and Techniques Conference (SIMUTools)*, 2009.

14. Bonaventura M, Wainer GA, and Castro R. Graphical modeling and simulation of discrete-event systems with CD++Builder. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(1):4–27, 2013.

15. Traoré MK. SimStudio: A next generation modeling and simulation framework. In *Proceedings of the International Simulation Tools and Techniques Conference (SIMUTools)*, 2008.

16. Quesnel G, Duboz R, and Ramat E. The Virtual Laboratory Environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17(4):641–653, 2009.

17. Franceschini R, Bisgambiglia PA, Touraille L, Bisgambiglia P, and Hill D. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *Proceedings of the Imperial College Computing Student Workshop (ICCSW)*, 2014.

18. Chow ACH and Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Proceedings of the Winter Simulation Conference (WSC)*, 1994.

19. de Lara J and Vangheluwe H. AToM3: A tool for multi-formalism and meta-modelling. In Kutsche RD and Weber H, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002.

20. Vangheluwe H, de Lara J, and Mosterman PJ. An introduction to multiparadigm modelling and simulation. In *Proceedings of the Simulation and Planning in High Autonomy Systems Conference (AIS)*, 2000.

21. Himmelspach J and Uhrmacher AM. Plug'n simulate. In *Proceedings of the Annual Simulation Symposium (ANSS)*, 2007.

22. Seo C, Zeigler BP, Coop R, and Kim D. DEVS modeling and simulation methodology with MS4 Me software tool. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2013.

23. Gamma E, Helm R, Johnson R, and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

24. Goldstein R, Breslav S, and Khan A. Using general modeling conventions for the shared development of building performance simulation software. In *Proceedings of the International Building Simulation Conference*, 2013.

25. Zimmermann G. A new approach to building simulation based on communicating objects.

26. Shneiderman B. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

27. Madlener F, Molter HG, and Huss SA. SC-DEVS: An efficient SystemC extension for the DEVS model of computation. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)*, 2009.

28. Barroca B, Mustafiz S, Van Mierlo S, and Vangheluwe H. Integrating a neutral action language in a devs modelling environment. In *Proceedings of the International Simulation Tools and Techniques Conference (SIMUTools)*, 2015.

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

30. Li X, Vangheluwe H, Lei Y, Song H, and Wang W. A testing framework for DEVS formalism implementations. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2011.

31. Nutaro J. adevs Documentation. Online API, 2013.

32. Wainer G. CD++: A toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

33. Hwang MH. *DEVS++: C++ open source library of DEVS formalism*, v.1.4.2 edition, 2009.

34. Goldstein R, Breslav S, and Khan A. Informal DEVS conventions motivated by practical considerations (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2013.

35. Meyers S. *Effective C++*. Addison-Wesley, Westford, MA, USA, 2005.

36. Maleki M, Woodbury R, Goldstein R, Breslav S, and Khan A. Designing DEVS visual interfaces for end-user programmers. *Simulation: Transactions of the Society for Modeling and Simulation International*, 91(8):715–734, 2015.

37. Davis AL and Keller RM. Data flow program graphs. *Computer*, 15(2):26–41, 1982.

38. Doore K, Vega D, and Fishwick P. A media-rich curriculum for modeling and simulation. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, 2015.

39. Varga A. *OMNeT++ User Manual (Version 4.6)*, 2014.

40. nsnam.org. ns-3.20 (Discrete-Event Network Simulator) documentation. Online API, 2014.

41. Goldstein R, Breslav S, and Khan A. A quantum of continuous simulated time. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2016.

42. Vicino D, Dalle O, and Wainer G. A data type for discretized time representation in devs. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, 2014.

43. Goldberg D. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

44. Hitz M, Werthner H, and Ören TI. Employing databases for large scale reuse of simulation models. In *Proceedings of the Winter Simulation Conference (WSC)*, 1993.

45. Breslav S, Goldstein R, Doherty B, Rumery D, and Khan A. Simulating the sensing of building occupancy. In *Proceedings of the Symposium on Simulation for Architecture and Urban Design (SimAUD)*, 2013.

46. Hunter JD. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

47. Goldstein R, Breslav S, and Khan A. Towards voxel-based algorithms for building performance simulation. In *Proceedings of the IBPSA-Canada eSim Conference*, 2014.

48. Autodesk Inc. Maya. Computer graphics software (www.autodesk.com/products/maya/overview-dts).

49. Gunay B, O'Brien L, Beausoleil-Morrison I, Goldstein R, Breslav S, and Khan A. Coupling stochastic occupant models to building performance simulation using the Discrete Event System Specification (DEVS) formalism. *Journal of Building Performance Simulation*, 7(6):457–478, 2014.

Author biographies

Rhys Goldstein is a Principal Research Scientist at Autodesk Research specializing in simulation theory and its application in architectural design. He received best paper awards at the IBPSA-USA 2010 and SpringSim 2016 conferences, and has served on the organizing committees of the SimAUD (Simulation for Architecture and Urban Design) and TMS/DEVS (Theory of Modeling and Simulation) symposia.

Simon Breslav is a Principal Research Scientist at Autodesk Research. Specializing in computer graphics, his current research interests include information visualization, simulation, and human-computer interaction.

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. Azam has published over 50 articles in simulation, human-computer interaction, architectural design, sensor networks, and sustainability.