DesignDEVS: Reinforcing Theoretical Principles in a Practical and Lightweight Simulation Environment

Rhys Goldstein, Simon Breslav, Azam Khan

Autodesk Research
Toronto, Canada
{rhys.goldstein, simon.breslav, azam.khan}@autodesk.com

ABSTRACT

We introduce DesignDEVS, a simulation development environment based on the Discrete Event System Specification (DEVS) formalism. DesignDEVS aims to promote understanding and appreciation of model-simulator separation, delayed binding of models, and other key principles of a systems engineering approach. To minimize installation and learning time, we embed a lightweight scripting language called Lua as the primary programming language for model implementation. Lua is extended to both enforce and communicate a number of modeling constraints implied by DEVS theory. These constraints include restrictions on state changes and data references. While not all theoretical principles are strictly enforced, we include a discussion of best practices which account for practical considerations such as modeler convenience and computational efficiency. DesignDEVS has been used for complex modeling tasks in architectural and building science research. Its unique features may aid in the teaching of DEVS.

Author Keywords

Simulation development environment; DEVS theory; Lua programming language; modeling constraints; best practices.

ACM Classification Keywords

I.6.7 Simulation and Modeling: Simulation Support Systems; I.6.5 Simulation and Modeling: Model Development.

1. INTRODUCTION

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 [9], block-diagram editing as realized by Simulink [19] and Ptolemy II [22], and traditional imperative programming as supported by C, C++, Java, Python, MATLAB, and even Fortran in many cases. An overarching goal of the field of modeling and simulation is to steer simulation developers—from all 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 a practitioner's time is limited. He/she is often inclined to stick with familiar programming techniques, instead of exploring alternatives that may or may not prove beneficial once all practical considerations are taken into account. Unfortunately, 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. Also, models often refer explicitly to one another, creating dependencies that discourage the testing of new combinations of models.

It is in this context that a formalism known as the Discrete Event System Specification (DEVS) [31] holds great promise. First, DEVS is among the most general of modeling formalisms. It has been shown that a multitude of other types of models can be formally mapped into DEVS models, though the reverse is not necessarily true [30]. 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. Overall, DEVS can be viewed as a means of delivering widespread benefit with moderate learning demands, while exposing practitioners to helpful principles such as model-simulator separation and delayed binding of models.

Numerous simulation development environments are available which provide both textual and graphical features for developing, debugging [20], and experimenting with simulation models. Among the simulation environments most dedicated to DEVS theory are those listed in Table 1: PowerDEVS [3], DEVS-Suite [32], CoSMoS [24], CD++ Builder [4], SimStudio [28], and VLE [23] (see [8] 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 [6], though various extensions may be supported as well. Each environment handles *atomic models* implemented in a textual programming language—usually C++ or Java—while some of the tools feature state-diagram-like editors that provide an alternative to imperative code. All of these tools offer a node-link diagram editor for defining *coupled models* that combine other models in a hierarchical fashion. 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 teach or reinforce 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.

SpringSim-TMS/DEVS 2016, April 3-6, 2016, Pasadena, CA, USA

Environment	DEVS Variant	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.

Some environments use DEVS, but feature it less prominently than those in Table 1. AToM^3 [7] exemplifies multi-paradigm modeling [29], where DEVS is regarded as a means of integrating models developed according to a diverse set of conventions. James II [14] also combines DEVS with other approaches. MS4 Me [25] 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. The tool introduced in this paper, *DesignDEVS*, has more in common with the simulation environments in Table 1. As illustrated in Section 2, elements of DEVS theory are prominently exhibited in the user interface. However, as with similar tools, the emphasis on DEVS is not meant to preclude support for modeling strategies seen as closer to the users' domains of expertise.

DesignDEVS aids in the teaching of DEVS principles with an emphasis on practical considerations such as ease of installation, rapid prototyping of models, and computational efficiency, to name of few. It was found that by embedding Lua into the environment as the primary programming language exposed to users, we could achieve the desired level of user convenience while furthering our educational objectives. Lua [16] is a minimalistic programming language with an interpreter specifically intended for embedding in software applications. This allows DesignDEVS to be distributed in a small and self-contained package, and eliminates the step of generating executable code from users' models. Lua is also a highly extensible language. Section 3 describes how Lua's environment tables can be modified to provide modeling-specific functions without importing large libraries. Furthermore, Lua's metatables can be exploited to prevent users from performing harmful operations. Explanations for these modeling constraints are communicated through error messages.

The most unique aspect of DesignDEVS is arguably the constraints it imposes on state changes and data references during run-time debugging. Consistent with DEVS theory, it is impossible to alter a state variable in the time advance function. More interestingly, if the state variable contains a data reference, then the value that is pointed to cannot change. The tool also features a novel solution to the "Insidious Pointer" problem as described by Nutaro [21]. In DesignDEVS, it is virtually impossible to communicate a data reference between two components in such a way that they alter one another's state by modifying the shared memory. Yet for the sake of efficiency, it is still possible to pass pointers under safe conditions as explained in Section 4. Whereas a number of DEVS-based libraries enforce some of these constraints using `const` variables or deep copies, the DesignDEVS approach is both comprehensive and informative. Any combination of value and pointer assignments that contradicts the formalism should produce a run-time error, and the associated restriction is explained to the user.

It is not practical to enforce all principles. Some principles must be promoted through best practices, as discussed in Section 5. An ongoing challenge is to establish best practices which address not only scalability but also user experience.

2. DESIGNDEVS USER INTERFACE

The layout of the DesignDEVS user interface, shown in Figure 1, reflects a key principle of modeling formalisms in general: the separation of model and simulator. The window is split by a horizontal divider bar into an upper section dedicated to modeling, and a lower section focused on simulation runs. All DEVS-based simulation environments feature this separation at a semantic level, but the visual partitioning of model and simulation elements reinforces the concept.

The modeling section above the horizontal divider is split into a model list on the left, and a model editor on the right. The editor is further organized by tabs, where the types of tabs depend on whether the selected model is atomic or coupled. If the selected model is coupled, as in Figure 1, there are 5 tabs including the Composition tab which contains the node-link diagram. If an atomic model is selected, the editor has 8 tabs including the Internal Transition tab shown in Figure 2 with Lua code for a classic Game of Life model.

The simulation section below the divider has a list of simulations on the left, and tabs for configuring and viewing the selected simulation on the right. These tabs include Messages, as shown in Figure 1, where input messages are created prior to running a simulation. The simulation end time is adjusted

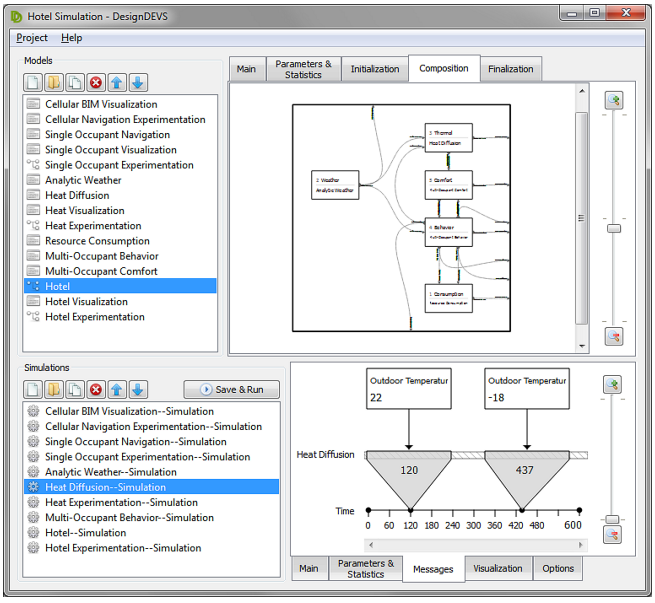

Figure 1. A screenshot of the DesignDEVS user interface showing a model list and editor (upper section), and a simulation list and configuration/visualization tools (lower section).

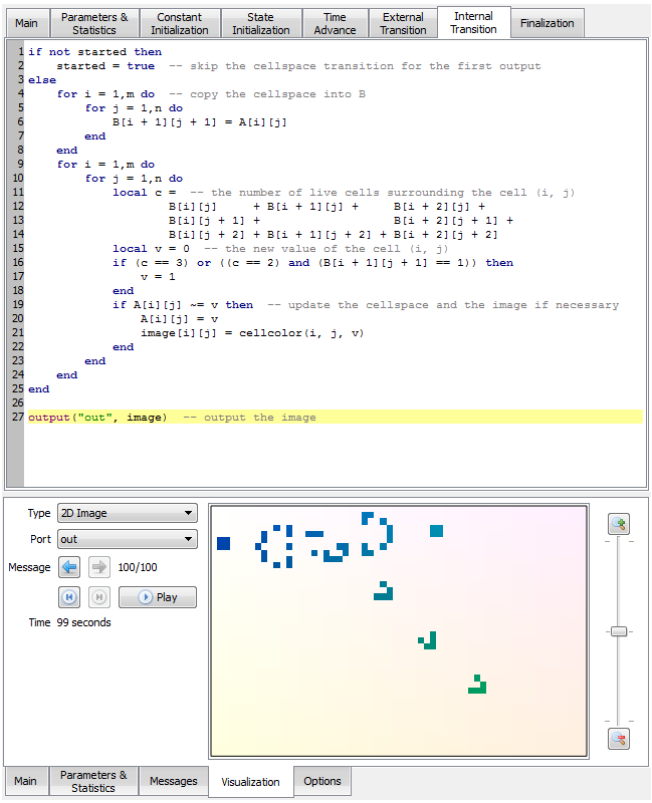

Figure 2. A transition function populated with Lua code for a classic Game of Life model (top), and a 2D visualization of the results (bottom).

on the far right of the timeline. Because items in the model list (top left of Figure 1) and the simulation list (bottom left) have a one-to-many relationship, multiple configurations can be stored for the same model. To support debugging after a simulation is run, both input and output messages are displayed on the timeline as seen in Figure 3 (output messages are lower and have upward arrows). Skewed triangular elements below messages remind users that discrete events are not necessarily evenly distributed over simulated time, and that multiple events can be associated with the same time point. As shown in Figure 2, there is also a Visualization tab for displaying 2D animations of simulation results.

The principle of 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.

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 4 and 5. Clicking on a blue stripe will expand the model vertically, as in Figure 4 (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 4 (bottom). Once a node is created, red text indicates that it needs a name and model. To prevent circular dependencies, 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 1. Nodes can be deleted or moved by clicking on either the red or blue square that appears at

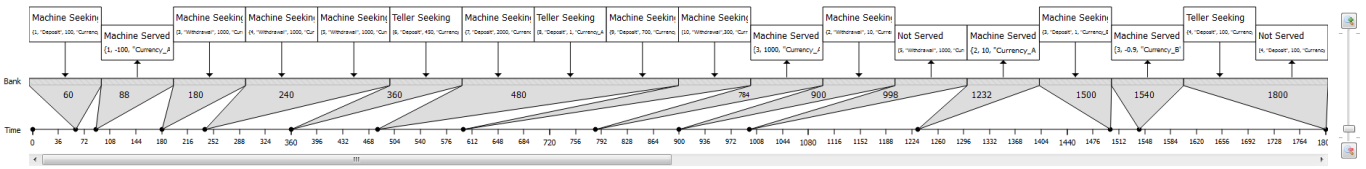

Figure 3. Pre-simulation input messages and post-simulation output messages, as appearing in the DesignDEVS timeline.

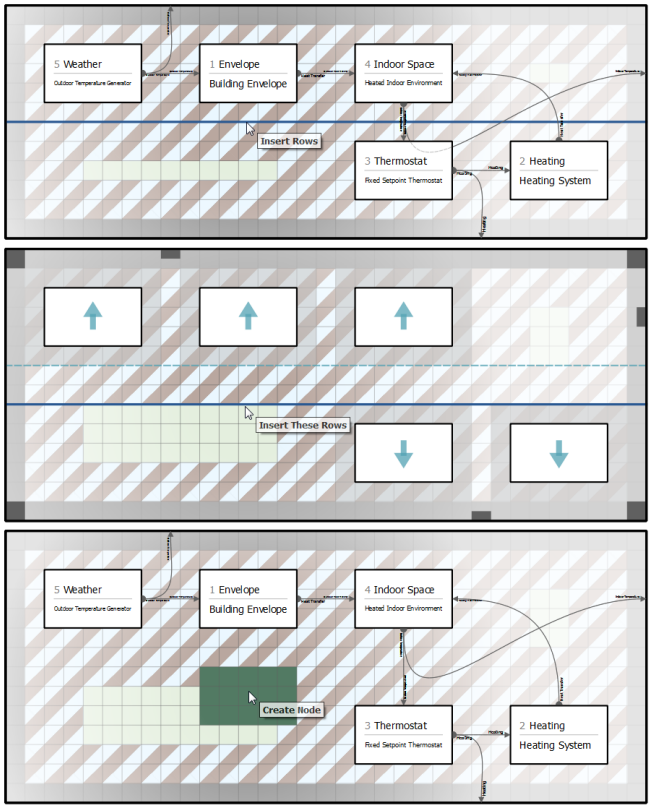

Figure 4. Coupled model expansion and node insertion.

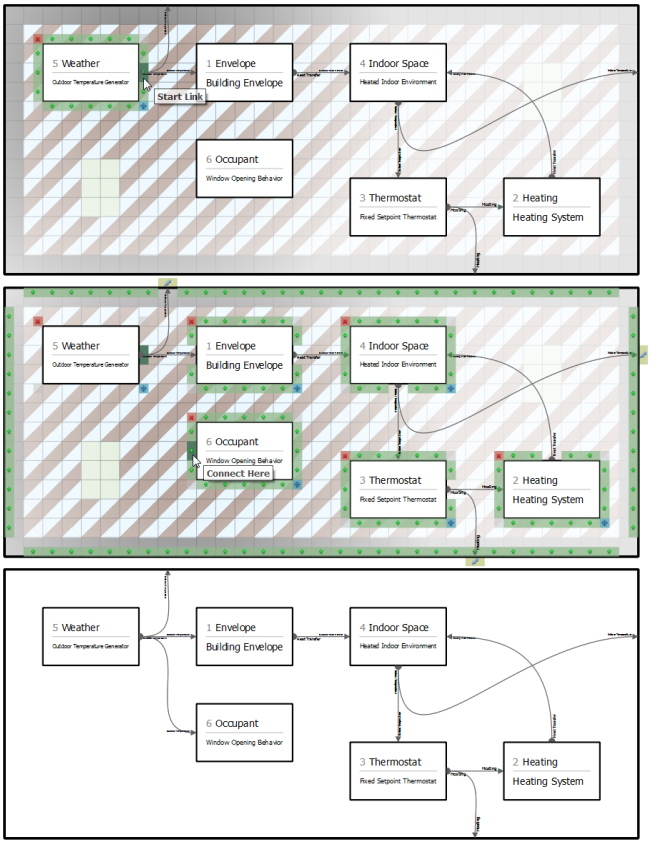

Figure 5. Coupled model link insertion.

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 5 illustrates the placement of a link and the final model as a result of the manipulations in both Figures 4 and 5.

DesignDEVS is developed for Windows and Mac OS using C++ with graphical user interface components from the Qt library [27], though only Lua is exposed to users.

3. LANGUAGE EMBEDDING AND EXTENDING

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 2. The `duration` function creates a time value from a multipler and a unit (e.g. `duration(8, "minutes")`). The value is rounded to the Time Resolution of the overall simulation, which is always as fine or finer than the Time Resolution of any of the component models. An optional third argument controls the rounding method: `"floor"`, `"ceil"`, `"halfup"` (default), `"halfdown"`, `"halfeven"`, or `"halfodd"`. With a single string argument, the `duration` function can produce a single unit of the simulation precision (`"minimum"`), the maximum representable duration (`"maximum"`), or an infinite duration (`"forever"`). A duration may appear anywhere in a DesignDEVS model, and in the Time Advance tab it serves as the `return` value. Other key functions include `input`, used in the External Transition tab to access received values; `output` used in the Internal Transition

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 2. Modeling-specific functions.

tab to send values; and `elapsed`, used where needed to obtain the time elapsed since the previous event. The `input` and `output` functions make use of the port names specified in the Main tab.

Environment table modifications allow models to be implemented with virtually no boilerplate code and no explicit importing of DEVS-specific libraries (though custom libraries can be imported with the `runscript` function). As a result, DEVS models can be rapidly prototyped. For example, Table 3 lists the source code for three basic models implemented in a mere 2, 4, or 18 lines. To help new users, these models and several others are included—along with step-by-step instructions contained in the model Description fields—in a Generator Processor Tutorial project packaged with the software. Also provided is the Game of Life model of Figure 2, and the coupled model example illustrated in Figures 4 and 5.

4. CONSTRAINT CHECKING AND COMMUNICATION

DEVS has a number of theoretical constraints by virtue of 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 constraints are enforced. SC-DEVS [18], 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 [2]. Nevertheless, few DEVS-based simulators fully protect the user from theoretical inconsistencies caused by the use of data references, or *pointers*.

As Nutaro [21] 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 problem is acknowledged in the testing framework of Li et al. [17], who investigate two simulators and find that both fail at least one related test case.

DesignDEVS promotes consistency with DEVS theory in large part through the detection of a broad range of errors at run-time. An example is the calling of a Table 2 function 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".

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

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 the wrong 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. The `const` feature of C++ does not necessarily provide this guarantee, since a `const` pointer does not mean the underlying data is constant. Furthermore, if we regard a model's entire state as a hierarchy (i.e. state variables may be tables containing other tables), a state table may only occur once in that hierarchy. 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"`). This modeling constraint, which also cannot be enforced using C++'s `const`, prevents 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*" [21]. 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"`). 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* which can be neither altered (`"attempt to modify a state variable owned by another model instance"`) nor stored (`"attempt to reference a modifiable table in state variables of multiple model instances"`).

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 in 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. By incorporating DEVS-specific error messages in a DEVS-inspired user interface, DesignDEVS removes many of the barriers novices encounter when developing formalism-based simulations.

5. APPLICATIONS AND BEST PRACTICES

DesignDEVS has served as the development environment for several simulation projects, including a model combining cognitive structural elements with motion sensors in buildings [5], and various explorations of equation-solving and time advancement strategies in the context of building system control and heat transfer [11, 13]. Although DesignDEVS can be applied to a multitude of domains, its name reflects an early focus on architectural design and building science.

Inevitably, the use of a simulation environment highlights practical considerations such as modeler convenience and computational efficiency, which are sometimes at odds with aspects of the underlying formalism [10]. In light of these considerations, we find that some DEVS principles are appropriate to enforce through constraints such as those in Section 4, whereas other principles can be encouraged by defining and communicating a set of best practices. An analogy can be made with object-oriented programming, where language designers tend to permit the declaration of public member variables despite recommending such variables be declared private. The overarching goal is to promote scalable methods while prioritizing user experience.

Consider the Classic DEVS output function λ. It is always invoked immediately before the internal transition function δ_{int}, but only the latter is permitted to modify the state. Suppose that a large array must be computed, output, and stored. Theoretically, either (*a*) the array must be computed first in λ

and then again in δ_{int}, or (*b*) λ and δ_{int} must both be invoked twice such that the first $\delta_{int}(s)$ occurs before the second $\lambda(s)$. This raises an unfortunate conflict between DEVS theory and the need for computational efficiency and convenience. To produce a positive experience for users, DesignDEVS shields them from this issue by merging λ and δ_{int} into a single Internal Transition tab where the state is permitted to change. An `output` function can be invoked any number of times from within the tab. The outputs are transmitted in sequence such that each is received by a separate External Transition event. Although this convention has theoretical drawbacks [10], there is precedent for the merging of λ and δ_{int} [15]. If one's purpose is to teach the DEVS formalism, a best practice could be established in which `output` is invoked at most once, with no preceding alteration of any state variable.

Many simulation environments allow users to directly modify the parameters of the components of a coupled model. DesignDEVS takes the opposite approach, enforcing the principle of encapsulation by giving every coupled model an Initialization tab. The modeler may read the parameters of the coupled model (using `get_parameter`) and produce 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. Atomic models are given an extra initialization step in the form of a Constant Initialization tab, where parameters are used to define a set of constants. These constants are actually mutable within the tab, but are automatically fixed prior to State Initialization.

DesignDEVS is similar to PowerDEVS in that models have user-supplied Finalization code. This code may be used to release access to resources, similar to destructors in C++. In DesignDEVS, they also accommodate the reporting of simulation results that have been aggregated into statistics. Not all users require statistics, but those that do benefit from a built-in reporting mechanism. The inclusion of statistics in models seems to contradict DEVS theory, which would confine them to an *experimental frame*. One can address this principle by recommending, as a best practice, that only experiment-specific *analysis models* be populated with statistics.

The most complex DesignDEVS model to date is a discrete-space hotel simulation [12]. A 3D animation of the results, produced separately in Autodesk Maya [1], is shown in Figure 6. 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 path-finding algorithms and other domain-specific modeling tasks.

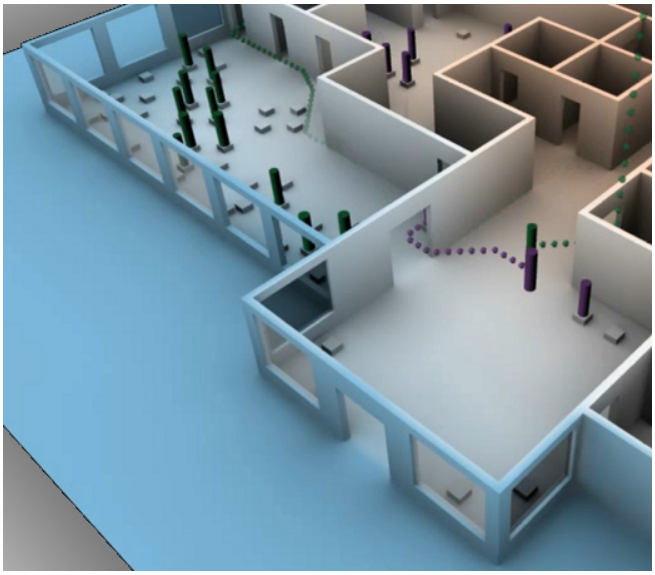

Figure 6. A virtual hotel modeled and simulated using DesignDEVS. The indoor temperature distribution (surface coloring) influences the window-opening behavior of occupants (cylinders), which in turn affects indoor temperature.

6. CONCLUSION

DesignDEVS has unique qualities that have not only proven effective for rapid prototyping, but will likely aid in the teaching of Classic DEVS. First, the simulation environment is distributed as a lightweight package that is easy to install. Second, the built-in scripting language simplifies the syntax of model code and avoids compilation-related difficulties. Most importantly, DesignDEVS reinforces its users' knowledge of DEVS theory through modeling constraints implemented via extensions to the Lua programming language, and communicated using formalism-specific error messages. By teaching the software alongside best practices, a broad range of theoretical principles can be covered. DesignDEVS contributes to the ongoing exploration of how to best incorporate modeling and simulation theory into practical tools. This will encourage practitioners from a multitude of disciplines to discover systems engineering principles on their own, helping them collaborate in the modeling of complex systems.

REFERENCES

1. Autodesk Inc. Maya. Proprietary animation software (www.autodesk.com/products/maya/overview-dts), 2016.

2. 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).

3. 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 (2011), 113–132.

4. Bonaventura, M., Wainer, G. A., 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 (2013), 4–27.

5. 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).
6. Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Proceedings of the Winter Simulation Conference (WSC)* (1994).
7. de Lara, J., and Vangheluwe, H. AToM3: A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering*, R.-D. Kutsche and H. Weber, Eds., vol. 2306 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002, 174–188.
8. Franceschini, R., Bisgambiglia, P.-A., 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).
9. 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).
10. 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).
11. 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).
12. Goldstein, R., Breslav, S., and Khan, A. Towards voxel-based algorithms for building performance simulation. In *Proceedings of the IBPSA-Canada eSim Conference* (2014).
13. 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 (2014), 457–478.
14. Himmelspach, J., and Uhrmacher, A. M. Plug'n simulate. In *Proceedings of the Annual Simulation Symposium (ANSS)* (2007).
15. Hwang, M. H. *DEVS++: C++ open source library of DEVS formalism*, v.1.4.2 ed., 2009.
16. Ierusalimschy, R., De Figueiredo, L. H., and Celes, W. Passing a language through the eye of a needle. *Communications of the ACM 54*, 7 (2011), 38–43.
17. 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).
18. Madlener, F., Molter, H. G., and Huss, S. A. 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).
19. MathWorks. *Simulink: Dynamic System Simulation for MATLAB*. 2000.
20. Mierlo, S. V., Tendeloo, Y. V., 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).
21. Nutaro, J. J. *Building Software for Simulation: Theory and Algorithms with Applications in C++*. John Wiley & Sons, Hoboken, NJ, USA, 2011.
22. Ptolemaeus, C., Ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
23. 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 (2009), 641–653.
24. Sarjoughian, H. S., 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).
25. Seo, C., Zeigler, B. P., 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).
26. Shneiderman, B. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology 1*, 3 (1982), 237–256.
27. The Qt Company Ltd. Qt Documentation. Online API (doc.qt.io/), 2016.
28. Traoré, M. K. SimStudio: A next generation modeling and simulation framework. In *Proceedings of the International Simulation Tools and Techniques Conference (SIMUTools)* (2008).
29. Vangheluwe, H., de Lara, J., and Mosterman, P. J. An introduction to multiparadigm modelling and simulation. In *Proceedings of the Simulation and Planning in High Autonomy Systems Conference (AIS)* (2000).
30. Vangheluwe, H. L. M. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design (CACSD)* (2000).
31. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, second ed. Academic Press, San Diego, CA, USA, 2000.
32. Zengin, A., and Sarjoughian, H. DEVS-Suite simulator: A tool teaching network protocols. In *Proceedings of the Winter Simulation Conference (WSC)* (2010).