

Introducing DEVS for Collaborative Building Simulation Development

Rhys Goldstein and Azam Khan
Autodesk Research
210 King Street East
Toronto, Ontario, Canada, M5A 1J7

September 30, 2010

Abstract

To accurately predict a building's energy use patterns, numerous interacting subsystems must be simulated in combination. These subsystems include indoor and outdoor climates, HVAC equipment, and occupant behavior, among others. We envision a collaborative simulation development process in which different researchers model different subsystems, then combine their work to produce realistic simulations of entire buildings. The problem is that models embedded in different simulation tools are rarely easy to integrate. In this tutorial we will introduce the audience to DEVS, and demonstrate how the formalism can be used to integrate models and encourage collaboration.

1 Introduction

Researchers and developers of *building performance simulation* (BPS) tools aim to provide architects with the decision support software they need to design energy-efficient buildings. Our interest lies in how these researchers and developers collaborate with one another. The need for collaboration is expressed well in [1]:

[Buildings are] becoming more and more heterogeneous with components from many engineering domains. They are complicated systems, governed by control laws, filled with innovative responsive elements (plant system, new material, strategic building features, etc.). Therefore, BPS tools must include these new complexities and be able to solve simultaneously coupled problems. [...] Due to this evolving and inherent interdisciplinary nature, collaboration among model developers should be certainly enabled and encouraged.

Buildings may be too complicated for any one developer to simulate with a high degree of accuracy. However, if several researchers each produce a realistic model of a specific aspect of a building, then entire buildings can be represented in considerable detail by combining the various models. One question then arises: how easy would it be to combine different models? Suppose that one researcher demonstrates an accurate model of indoor air temperature flow, and another produces a state-of-the-art model of occupant behavior. How easy would it be to link the two models, such that temperature changes from the first model affect human behavior produced by the second?

If researchers adopt a traditional approach to simulation development, we would expect their models to be very difficult to extract and combine. Firstly, their *model* code, the code describing real-world systems, would in all likelihood be intertwined with their *simulator* code, the code instructing the computer on how to perform a simulation. Secondly, assuming all the model code could somehow be separated from the simulator code, we would not expect there to be an obvious way to connect the various models together.

What is needed is an alternative to the traditional approach to simulation development: a common strategy to be adopted by numerous building simulation researchers from different institutions. We recommend the *Discrete Event Systems Specification* (DEVS), a formalism established to model a wide range of systems that change over time.

The purpose of this paper is to introduce DEVS to the BPS community, and propose that the formalism be adopted to enable collaboration between building simulation researchers. We start with a brief overview of simulation use and simulation development (Section 2), followed by an example demonstrating the development of a simple building simulation using a traditional approach (Section 3). We then introduce DEVS (Section 4), and demonstrate its use by applying it to the same building simulation (Section 5). Before concluding, we explain how DEVS fits in with object-oriented programming and other modeling formalisms (Section 6).

2 Simulation Use and Simulation Development

A simple illustration of simulation use is shown in Figure 1. Input data, which consists primarily of *model parameters* pertaining to a real-world system, is supplied to simulation software. The software then outputs simulation results.



Figure 1: A basic illustration of simulation use.

Looking at Figure 1, it appears as though the simulation results are the user’s primary objective. But more often than not, the user spends their time pursuing an optimal set of model parameters using the iterative process shown in Figure 2. After the user runs the first simulation, the results obtained will likely inspire changes to the model parameters. The new set of parameters then produces new simulation results, which inspires further changes. The user’s primary objective is the final set of model parameters, as indicated by the red circle in Figure 2.

In the case of a building simulation, we refer to a set of model parameters as a *building information model* (BIM). An architect may begin a new project by designing a building envelope and producing a very simple BIM. He/she might then supply that BIM to a simulation tool, and the simulation results would predict the amount of energy the building

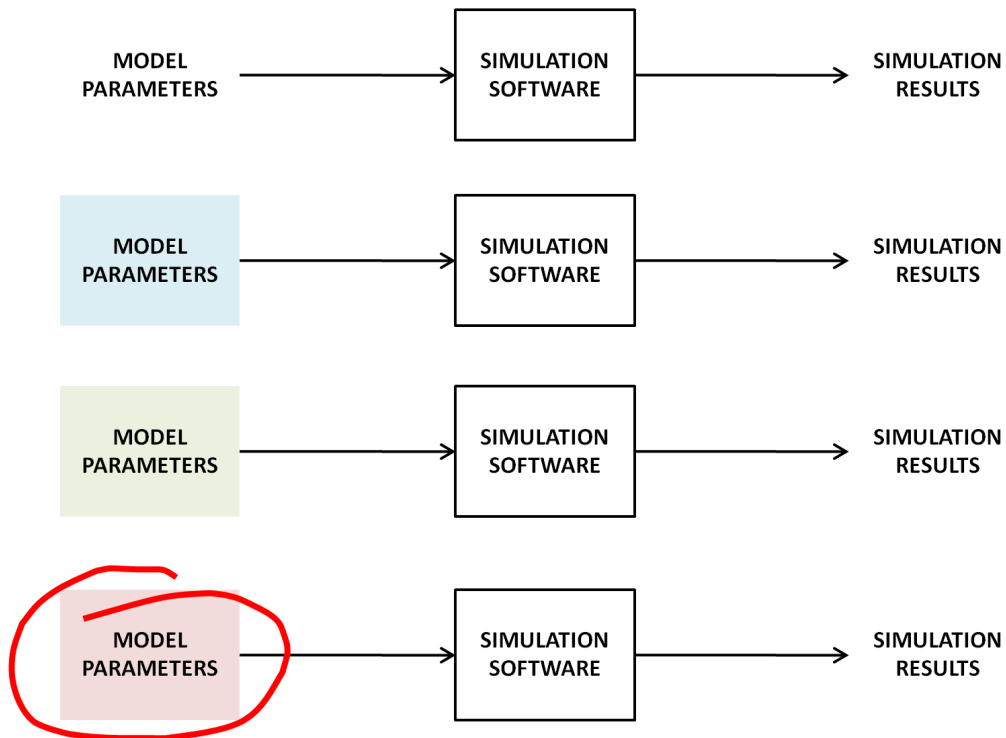


Figure 2: Simulation use following an iterative process.

would consume. The architect may then add interior walls to the BIM and run a new simulation. As the project progresses, various design decisions are informed by the simulation results, but the architect’s deliverable is the final BIM (the final set of model parameters).

Turning our attention from simulation use to simulation development, we note that a developer follows an analogous iterative process. Again, as illustrated in Figure 3, simulations are performed many times over. In the case of simulation development, it is the simulation software that is enhanced with each iteration. The software developer’s objective is, of course, a deliverable version of the software.

Ideally, a researcher developing BPS software would have on hand a BIM to use as a set of model parameters, and a real building represented by that BIM. He/she would start with computer code implementing very simple simulation algorithms, supply the BIM, and obtain simulation results. If the results obtained were to differ significantly from corresponding measurements collected within the real building, the researcher would enhance the simulation algorithms, run a new simulation, and continue repeating this process.

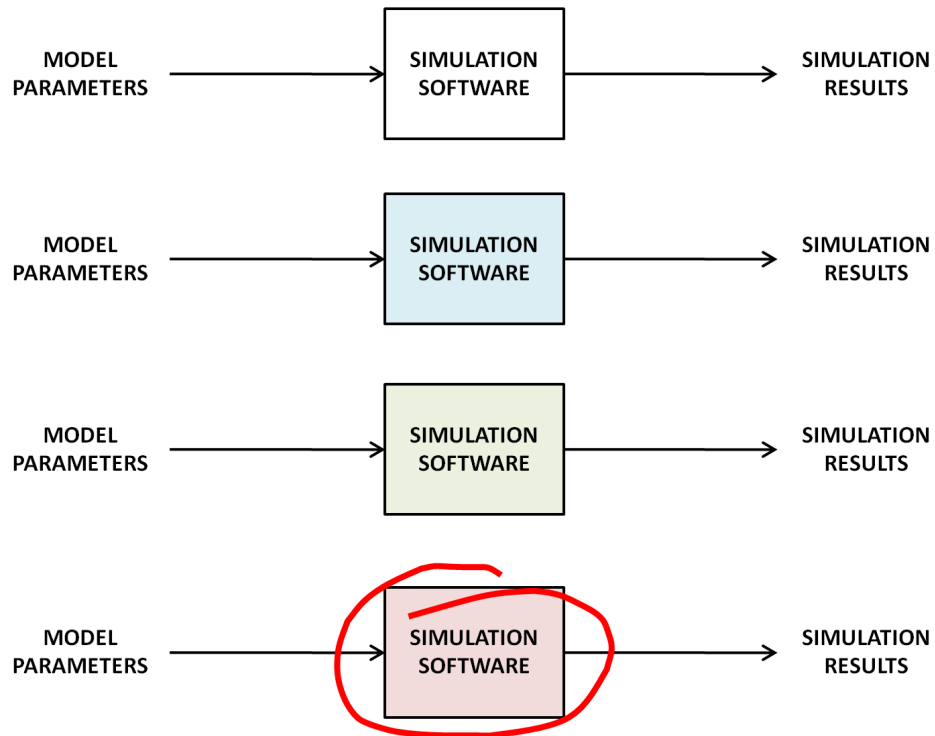


Figure 3: Simulation development following an iterative process.

Our focus in this paper will be on simulation development as an iterative process. We do not regard a building simulation as a static entity, but rather as something that continually changes. Accordingly, we concern ourselves relatively little with the complexity of simulation code as it exists at any one point in time, but we are very concerned with how easy simulation code is to modify and enhance.

Key Points:

- Simulation use and simulation development are both iterative processes.
- Simulation users run simulations repeatedly, improving model parameters with each iteration.
- Simulation developers run simulations repeatedly, enhancing simulation software with each iteration.

3 An Example of Traditional Simulation Development

Here we demonstrate a traditional approach to simulation development using a simple building simulation as an example.

South Dakota State University provides free access to archived weather data. We used their online service to obtain hourly outdoor temperature, and with that data developed the first version of our simple building simulation. At this stage, the building itself is not modeled. Our “outdoor climate simulation” does nothing more than output the sequence of outdoor temperatures collected for Iowa City over a two-week period in the summer of 2009. The simulation was implemented using the Python programming language. The code is listed in Figure 4.

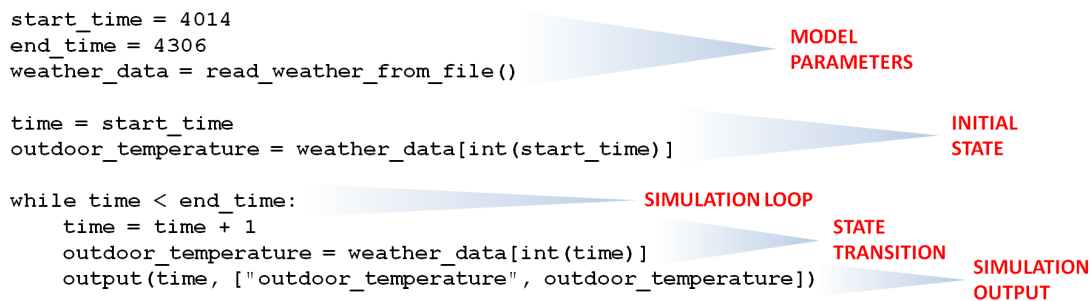


Figure 4: Code for an outdoor climate simulation.

We have omitted implementations of the functions `read_weather_from_file` and `output`, which access files to read or write information. Otherwise, Figure 4 gives the complete simulation program. Note that the code consists of five parts: the model parameters, which are normally supplied by the user; the initialization of a set of changing variables known as the *state*; a simulation loop; the state transition, which occurs repeatedly within the loop; and the simulation output, which in this case also occurs within the loop.

There is nothing particularly interesting about this simulation. Remember that our focus is on the process through which simulation code is enhanced, which we will examine shortly. For now, note that all simulations tend to include the five aspects listed above. Also observe the simulation results shown in Figure 5. The outdoor temperature fluctuates between roughly 60 and 90 degrees Fahrenheit over a 14-day period.

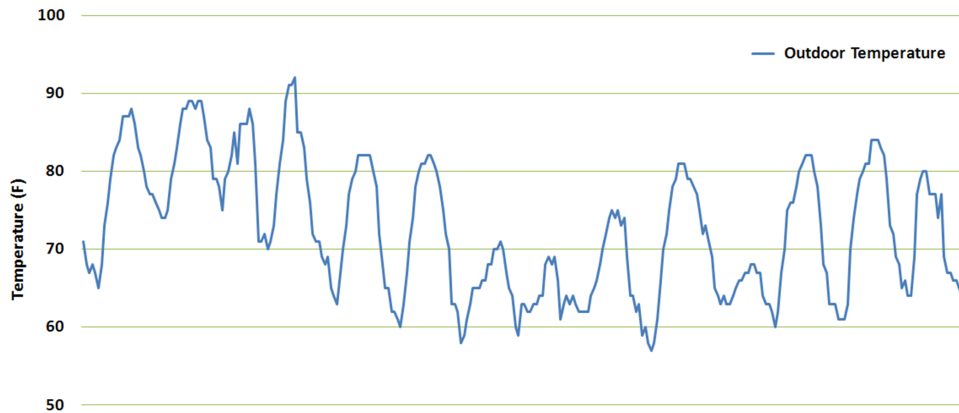


Figure 5: Outdoor climate simulation results.

Simulation development is an iterative process; now that we have a first version of our software and have used it to obtain simulation results, we should enhance the software. We therefore add a simple building model that consists of a building envelope and an interior space with a uniform indoor temperature. We use Newton’s law of cooling to model the transfer of heat from the outdoor environment to the building’s interior. The code for this simple “indoor climate simulation” is listed in Figure 6.

```

start_time = 4014
end_time = 4306
weather_data = read_weather_from_file()
wall_rate = 0.1

time = start_time
outdoor_temperature = weather_data[int(start_time)]
indoor_temperature = weather_data[int(start_time)]
lower_transition_temperature = indoor_temperature - 1.0
upper_transition_temperature = indoor_temperature + 1.0

while time < end_time:
    outdoor_transition_time = int(time) + 1

    rate = wall_rate
    target_temperature = outdoor_temperature
    dt = target_temperature - indoor_temperature

    if dt < 0:
        transition_dt = lower_transition_temperature - indoor_temperature
    else:
        transition_dt = upper_transition_temperature - indoor_temperature
    if abs(dt) <= abs(transition_dt):
        indoor_transition_time = infity
    else:
        indoor_transition_time = time + (1.0/rate)*log(abs(dt)/(abs(dt) - abs(transition_dt)))

    if indoor_transition_time < outdoor_transition_time:
        if dt < 0:
            indoor_temperature = lower_transition_temperature
        else:
            indoor_temperature = upper_transition_temperature
        lower_transition_temperature = indoor_temperature - 1.0
        upper_transition_temperature = indoor_temperature + 1.0
        time = indoor_transition_time
    else:
        dt = outdoor_transition_time - time
        indoor_temperature = target_temperature - dt*exp(-rate*dt)
        time = outdoor_transition_time
        outdoor_temperature = weather_data[int(time)]
        output(time, ["outdoor_temperature", outdoor_temperature])
        output(time, ["indoor_temperature", indoor_temperature])

```

MODEL
PARAMETERS

INITIAL
STATE

SIMULATION LOOP

STATE
TRANSITION

SIMULATION
OUTPUT

Figure 6: Code for an indoor climate simulation.

Do not bother trying to understand the code in Figure 6, but make two observations. First note that the code can still be partitioned into the same five parts mentioned earlier. We have added an extra model parameter to capture the rate at which heat is transferred through the walls of the building, the simulation state now includes 5 variables instead of 2, and the state transition code is considerably longer. However, the basic structure of the program remains. The second thing to note is that the added code, shown in green, is scattered throughout the program.

As shown in Figure 7, the simulation results now include both outdoor and indoor temperatures. The indoor temperature lags the outdoor temperature, but captures the same prominent rises and falls.

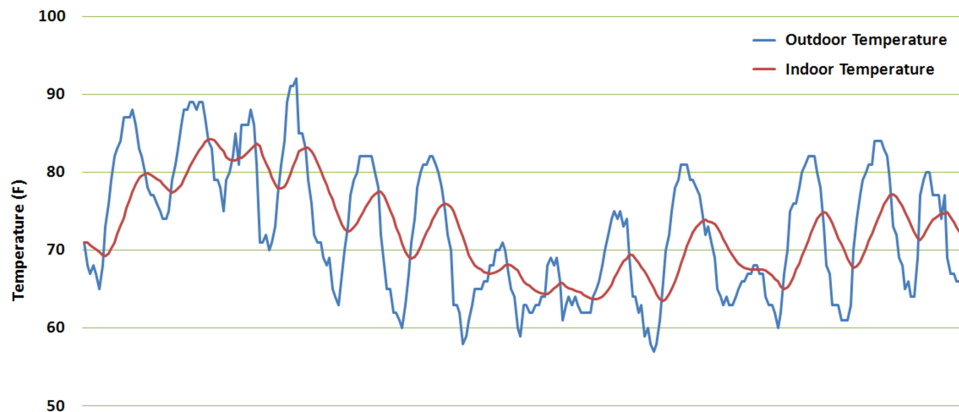


Figure 7: Indoor climate simulation results.

We now add a heating system to our simulation. If the indoor temperature drops to 70 degrees Fahrenheit, a heater will turn on and supply heat at a relatively fast rate. When the indoor temperature reaches 75 degrees, the heater will turn off. Figure 8 shows both the code and the simulation results for this “heating system simulation”. Note that halfway through the simulation, the indoor temperature exhibits a sawtooth pattern as a result of the low outdoor temperature and the intermittent source of heat.

For the fourth and final iteration, we add an occupant to the model. If the indoor temperature is at least 76 degrees, and the indoor temperature is higher than the outdoor temperature, the occupant opens a window. This increases the rate at which heat is transferred through the building envelope, accelerating rate at which building cools down. If the in-

```

start_time = 0
end_time = 24
initial_time = 0
initial_temperature = 70
initial_heating_system_status = 0
initial_window_status = 0
initial_occupant_status = 0
initial_heating_system_temperature = 70
initial_occupant_temperature = 70
initial_heating_system_capacity = 10
initial_occupant_capacity = 10

time = 0
while time < end_time:
    # Heating System
    heating_system_temperature = max(heating_system_temperature - 1, 0)
    heating_system_capacity = max(heating_system_capacity - 1, 0)
    heating_system_status = 1 if heating_system_temperature < 70 else 0

    # Occupant
    occupant_temperature = max(occupant_temperature - 1, 0)
    occupant_capacity = max(occupant_capacity - 1, 0)
    occupant_status = 1 if occupant_temperature < 72 else 0

    # Window
    window_status = 1 if occupant_status == 1 else 0

    # Heating System
    heating_system_temperature = max(heating_system_temperature + 1, 0)
    heating_system_capacity = max(heating_system_capacity + 1, 0)
    heating_system_status = 1 if heating_system_temperature < 70 else 0

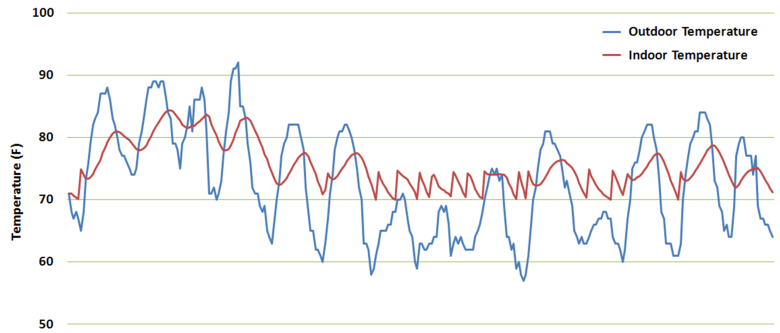
    # Occupant
    occupant_temperature = max(occupant_temperature + 1, 0)
    occupant_capacity = max(occupant_capacity + 1, 0)
    occupant_status = 1 if occupant_temperature < 72 else 0

    # Window
    window_status = 1 if occupant_status == 1 else 0

    time = time + 1

```

(a) Code



(b) Results

Figure 8: Heating system simulation.

door temperature then reaches 72 degrees, the occupant closes the window. The code and results of this “window opening simulation” are shown in Figure 9. The increased rate of cooling can be observed by comparing these results (Figure 9b) with those of the simulation with the heating system but no occupant (Figure 8b).

```

start_time = 0
end_time = 24
initial_time = 0
initial_temperature = 70
initial_heating_system_status = 0
initial_window_status = 0
initial_occupant_status = 0
initial_heating_system_temperature = 70
initial_occupant_temperature = 70
initial_heating_system_capacity = 10
initial_occupant_capacity = 10

time = 0
while time < end_time:
    # Heating System
    heating_system_temperature = max(heating_system_temperature - 1, 0)
    heating_system_capacity = max(heating_system_capacity - 1, 0)
    heating_system_status = 1 if heating_system_temperature < 70 else 0

    # Occupant
    occupant_temperature = max(occupant_temperature - 1, 0)
    occupant_capacity = max(occupant_capacity - 1, 0)
    occupant_status = 1 if occupant_temperature < 72 else 0

    # Window
    window_status = 1 if occupant_status == 1 else 0

    # Heating System
    heating_system_temperature = max(heating_system_temperature + 1, 0)
    heating_system_capacity = max(heating_system_capacity + 1, 0)
    heating_system_status = 1 if heating_system_temperature < 70 else 0

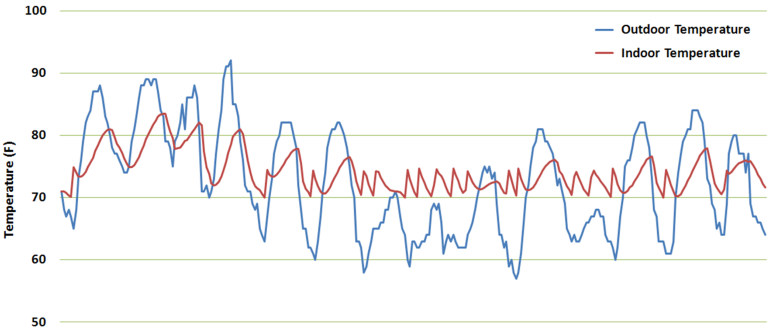
    # Occupant
    occupant_temperature = max(occupant_temperature + 1, 0)
    occupant_capacity = max(occupant_capacity + 1, 0)
    occupant_status = 1 if occupant_temperature < 72 else 0

    # Window
    window_status = 1 if occupant_status == 1 else 0

    time = time + 1

```

(a) Code



(b) Results

Figure 9: Window opening simulation.

When we added the heating system to our simulation, we inserted the code shown in green in Figure 8a. Similarly, the green code in Figure 9a was inserted to incorporate the occupant and window in the subsequent version. The key thing to note is that in both cases, changes were made throughout the program.

We assert that the need to repeatedly modify simulation code in numerous locations discourages collaboration. Suppose that a developer wanted to add an airconditioning system to our example simulation. Because the necessary modifications could be located anywhere

in the code, the developer must understand the program in its entirety before proceeding. Suppose also that a researcher wanted to modify the simulation such that the occupant is absent during certain hours of the day, and therefore unable to manipulate the window. If the developer and the researcher each take the Figure 9a code and make their changes independently, it would be very difficult to combine their enhancements and obtain a simulation with an air conditioner and periods of absence.

The problem only worsens as we transition from a “toy” simulation, like the one presented here, to a fully-featured whole building simulation. In this example, the model parameters consisted of a few variables and a text file with hourly temperature values. A serious effort to predict the energy consumption patterns of a real building will require a simulation in which the model parameters consist of an entire BIM, and a BIM may contain 100 Megabytes of data or more. If the code dealing with model parameters is scaled from the first few lines of Figure 9a to an entire BIM, one can only imagine the resulting quantity of simulation code. Should multiple developers want to contribute to such a large and complex code base, an alternative to the traditional approach is necessary.

Key Points:

- Simulation code generally deals with model parameters, the initial state of a model, a simulation loop, state transitions within the loop, and simulation output.
- Following a traditional approach to simulation development, each enhancement requires changes to be made throughout the code; this discourages collaboration.

4 DEVS-Based Simulation Development

DEVS it is a set of conventions. It is not a programming language, and it is not a specific software application. One can develop a DEVS-based simulation using any one of a number of existing DEVS-based tools, or one can program an entire DEVS-based simulation from scratch using any programming language. Either way, so long as a certain set of conventions are followed, one is using DEVS.

DEVS was invented by Bernard Zeigler, who introduced it with his 1976 book on modeling and simulation [2]. When we refer to DEVS in this paper, we are referring to the DEVS formalism: the set of conventions mentioned above. It is worth noting that DEVS was developed hand-in-hand with a theory. Zeigler discusses his theory in [3]:

[...] there are large research communities that could be considered to be doing modeling and simulation, who do not necessarily feel the need for a theory of Modeling and Simulation, let alone accept mine. For example, people working in computational science are concerned with making complex scientific codes run faster typically using high performance computers. They tend to blur the distinction between model and simulator, seeing only code that can be partitioned and instrumented.

The first thing to remember about DEVS is that it forces developers to make a clear distinction between models and simulators. Recall that in Figure 1, we illustrated the basic simulation procedure as the use of simulation software to input model parameters and produce simulation results. With DEVS, the diagram in Figure 10 is more appropriate. Here the simulation software is partitioned into a model and a simulator. The model parameters customize a model, which is supplied to a simulator, which produces the results.

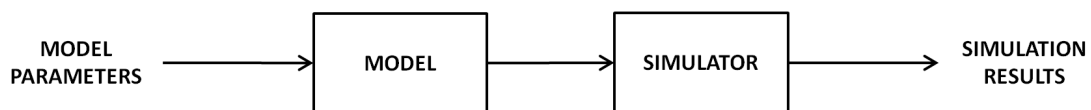


Figure 10: DEVS-based simulation.

Even without knowing the difference between the two, one can tell that the simulation programs presented in Section 3 “blur the distinction between model and simulator”. In each version of that example, there was only a single chunk of code. In general, a *model* is a description of a system. A *DEVS model* is a specific type of model in which the description is composed of mathematical formulas (or computer code), and those formulas (or code) are structured in a particular way. A *simulator*, by contrast, is whatever interprets a model and performs the simulation. A *DEVS simulator* is a computer program that defines a process in which simulated time is repeatedly advanced and the formulas in a DEVS model are repeatedly evaluated. Once implemented, a DEVS simulator should work for a wide

range of DEVS models, regardless of whether the models represent buildings, animals, robots, or any other type of system.

Just like the traditional approach, DEVS-based simulation development is an iterative process. The difference is that, because a DEVS simulator can be reused for a variety of different models, the developer modifies only the model at each iteration. As shown in Figure 11, a realistic model is the objective.

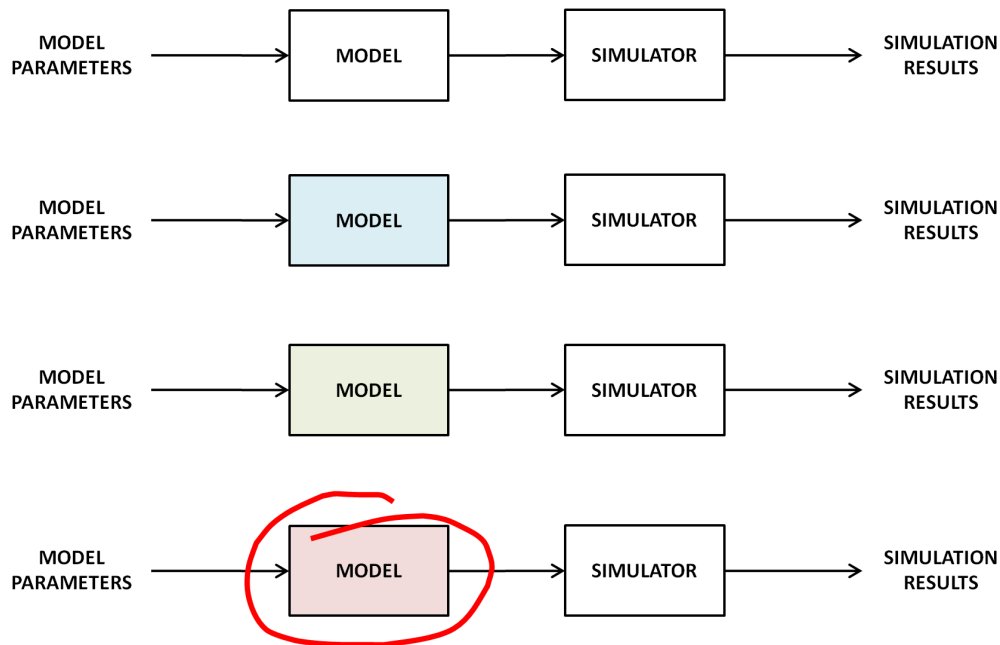


Figure 11: DEVS-based simulation development following an iterative process.

Aside from enforcing a distinction between the model and the simulator, DEVS allows complex models to be composed of simpler submodels. We will demonstrate this property in the next section by approaching our simple building simulation with DEVS.

Key Points:

- Using DEVS, simulation software is divided into a model and a simulator.
- With a DEVS simulator in hand, a developer runs simulations repeatedly and enhances the DEVS model with each iteration.

5 An Example of DEVS-Based Simulation Development

Before we proceed with the example, we should clarify what we mean by “model” in the context of BPS. Architects and engineers use the word to describe a building design, as in “building information *model*”. In a simulation context, however, we regard a BIM as a set of model parameters. What we refer to as a “model” is generally a description of a process that has some noteworthy influence on a building. To us, the size and shape of a building are parameters, but a computational fluid dynamics algorithm that predicts indoor air flow is a model.

As in Section 3, we begin with a simulation of outdoor climate. Instead of worrying about the entire “outdoor climate simulation”, however, we need only define an “outdoor climate model”. The simulator component of the software is reusable, and we will assume we had one on hand from the outset. The outdoor climate model has one output, illustrated in Figure 12 with an exiting arrow. Had the model included inputs as well, we would have drawn an additional arrow pointing into the model.

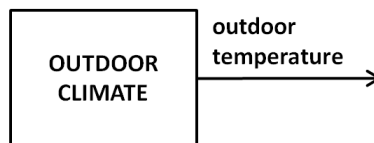


Figure 12: Diagram of a DEVS outdoor climate model.

DEVS models are generally classified as either atomic or coupled. An *atomic model* is indivisible, whereas a *coupled model* is composed of other DEVS models. Because the outdoor climate model of Figure 12 is atomic, DEVS requires us to define it using a specific set of functions: an `external_transition` function, an `internal_transition` function, and a `time_advance` function. As shown in Figure 13, these three functions in combination are assigned to the variable `DEVS_model`. The example also includes an initialization function called `initialize`, a worthwhile addition to any DEVS model.

A couple things are worth noting. First, it is common practice to use four functions instead of three in a DEVS model, plus three additional variables representing sets of inputs, outputs, and states. For the sake of simplicity, we present a very minimalistic formulation of DEVS. Second, because simplicity was our main concern in this case, we chose to imple-

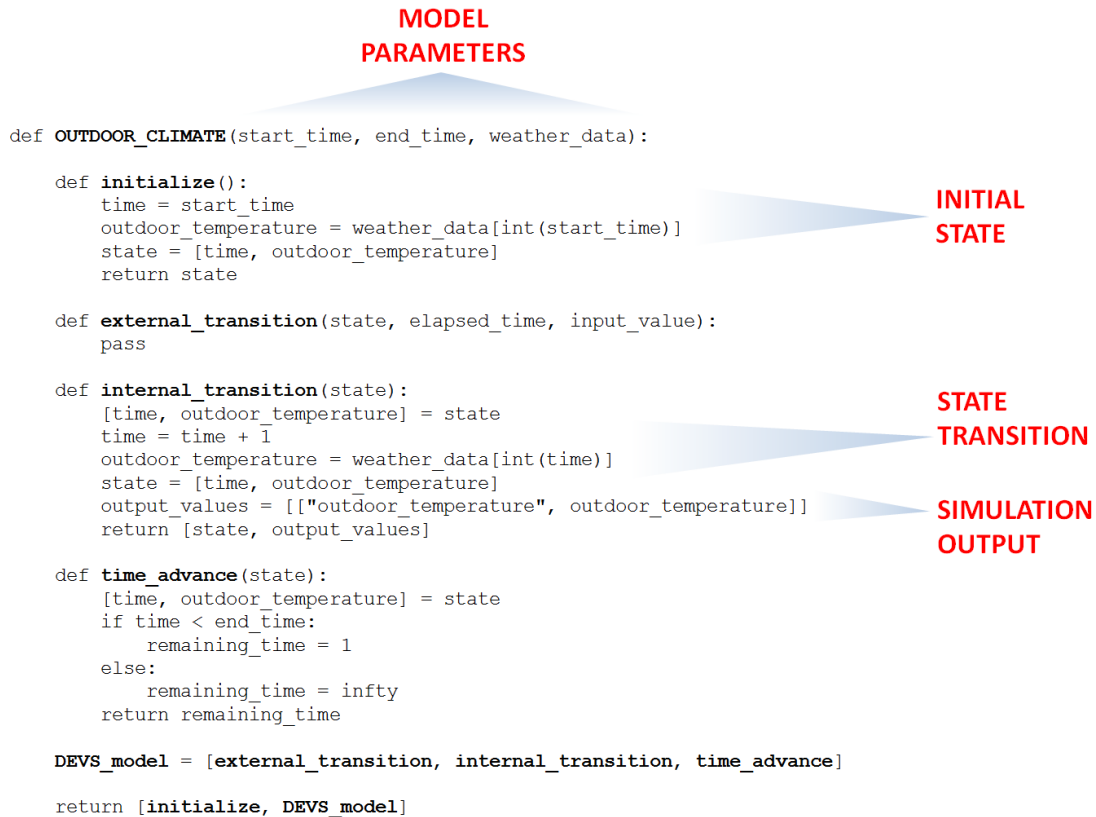


Figure 13: DEVS outdoor climate model code.

ment these examples in Python. The DEVS simulator that invokes the functions shown is described in detail in [4]. Remember that DEVS can be written in any language, however; had performance been our primary concern, we might have chosen a C++ DEVS simulator like the one described in [5].

The code in Figure 13, when supplied to the simulator, produces exactly the same results as the non-DEVS simulation code in Figure 4. Much of the code is the same as well. The DEVS model features the same three model parameters (though the values are omitted), identical state initialization and state transition code, and very similar simulation output code. The simulation loop is excluded from the DEVS model, as it is part of the simulator. One should not be discouraged by the fact that the DEVS model requires more code than the corresponding traditional version. It is true that there is some overhead associated with DEVS. In practice, however, models designed for BPS will be far more complex than those in this paper, and the code associated with DEVS will end up being only a small fraction

of the software. Remember that our interest lies in the process by which a simulation is enhanced. As we add to our outdoor climate model, the benefits of DEVS will become apparent.

As before, our first enhancement is the addition of a building envelope enclosing an indoor space. This requires us to define a “building envelope model” and an “indoor climate model”. By linking these two atomic models together, along with the previous atomic model, we obtain the coupled model shown in Figure 14.

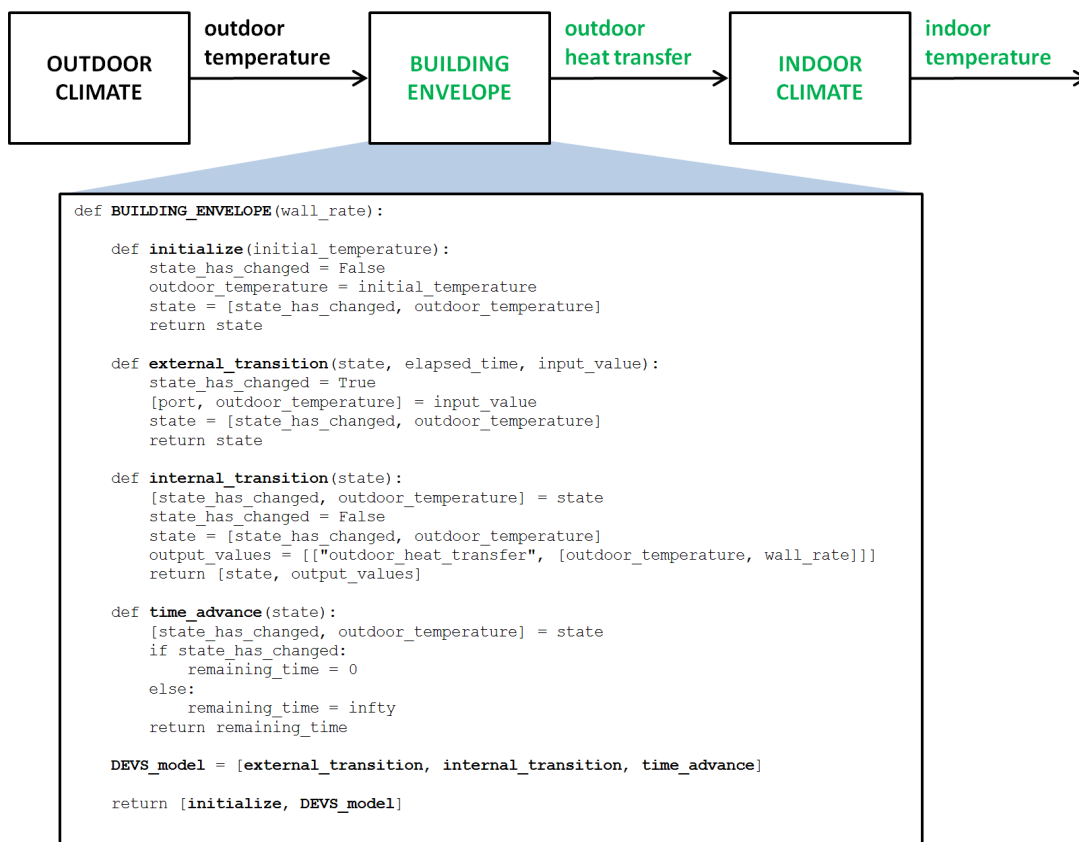


Figure 14: A DEVS indoor climate model within a DEVS coupled model.

Recall that the outdoor climate model had an output but no inputs (Figure 12). This is reflected in the code (Figure 13), as its `internal_transition` function contained state transition code, while the `external_transition` function was basically empty (pass means “do nothing”). The building envelope model, by contrast, has both an input

and an output (Figure 14). Accordingly, both of its transition functions contain state transition code. Note that the output of the outdoor climate model becomes the input of the building envelope model.

With this first enhancement, the outdoor climate model implemented in Figure 13 remained completely unchanged. And as is generally the case with DEVS, the simulator remained unchanged as well. The additional work included the creation of two new DEVS atomic models, and the coupling of three models. In this case, the use of DEVS made it easy to distinguish the parts of the overall program that required modification from the parts that could be left alone.

Recall that the next enhancement was the addition of a heating system activated at an indoor temperature of 70 degrees Fahrenheit, and deactivated at 75 degrees. Using DEVS, we add a “sensor model” and a “heater model”. As shown in Figure 15, the output of the indoor climate model becomes the input of the sensor model. The output of the heater model also becomes a second input to the indoor climate model, and consequently the indoor climate model must be modified. So two atomic models remain the same, two are added, and one is modified. Again, the changes were relatively localized and easy to identify.

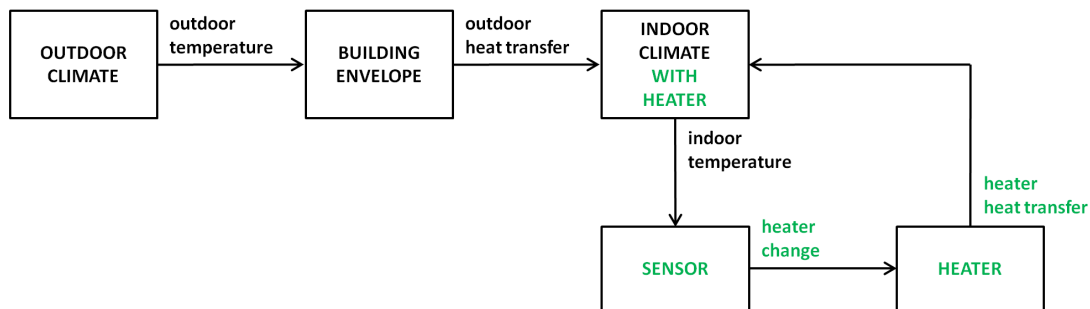


Figure 15: The coupled model with a sensor, a heater, and a modified indoor climate model.

The final stage of our simulation includes an occupant who manipulates a window in the building envelope. We now add an “occupant model” and modify the building envelope model. The final DEVS coupled model, shown in Figure 16, produces results matching those of the last version of the code developed via the traditional approach. Using DEVS, this last enhancement required one additional atomic model, and modifications to one existing model. The majority of the existing submodels were not changed.

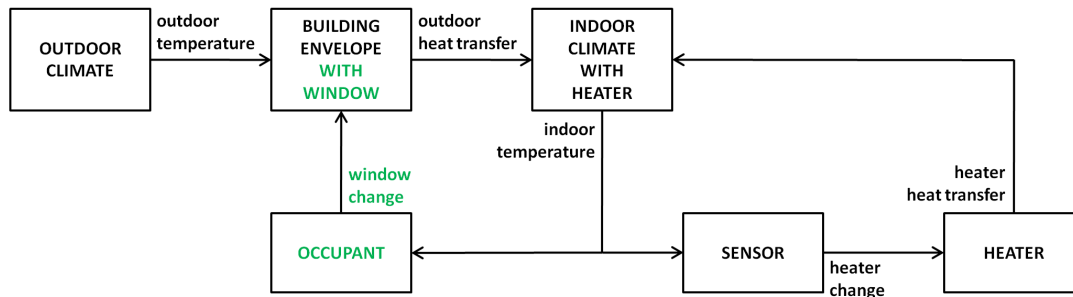


Figure 16: The coupled model with a occupant and a modified building envelope.

We speculated earlier that if a developer wanted to add an air conditioning system to our simulation, without DEVS, he/she could expect to make modifications throughout the code. With DEVS, a possible set of modifications are easy to envision: another sensor model, which would send information to a new air conditioning model, which in turn would affect heat exchanged with a modified indoor climate model. Suppose that as the developer was making those modifications, another researcher was enhancing the occupant model to include periods of absence. Following the traditional approach, it is unclear how the two contributors would combine their work. With DEVS, however, it is quite possible that the enhancements could be integrated by simply selecting the latest versions of each individual submodel, and linking them together.

Obviously DEVS would not solve all collaboration-related issues. If two developers independently modified the same atomic model, then merging their changes would involve more work than merely connecting different submodels together. But at least they would know that the simulator and majority of the submodels could be left alone.

Key Points:

- A DEVS model generally deals with model parameters, the initial state, state transitions, and simulation output, but the simulation loop itself is part of the simulator.
- DEVS models are either atomic or coupled, and coupled models include various interconnected submodels.
- Using DEVS, each enhancement requires modifications to only certain submodels; this encourages collaboration.

6 Other Formalisms and Techniques

DEVS can be described as the “common denominator” of modeling formalisms [6]. Models designed using Petri Nets, Finite State Automata, State Charts, and many other formalisms can be expressed with DEVS, though the reverse is not always true. Despite its generality, DEVS is not particularly complicated. A DEVS model is either a collection of other models or, at its simplest, a set of only three functions. For these reasons, we recommend DEVS over all other modeling formalisms.

One might argue that the advantages we attributed to DEVS could have been derived from well-known programming techniques, functions and object-oriented features being the obvious examples. First we would like to stress that the adoption of DEVS does not prevent the use of functions or objects. That said, we would argue that functions and objects alone are not enough. If we took the code in Section 3 and re-organized it into various functions, we may well succeed in making the code easier to understand. However, the mere introduction of functions gives us no reason to believe that future changes will be localized. Each enhancement would likely require modifications to numerous functions.

The principles behind object-oriented programming are quite similar to those underlying DEVS, and one could certainly use a separate object for each submodel shown in our example of DEVS-based simulation development. If a developer were to use objects with no knowledge of DEVS, however, it is unlikely that he/she would end up keeping the model and simulator separate. Another feature of DEVS is that all state transitions may depend on the simulated time in one way or another. With a quick look at back at Figures 13 and 14, one will observe that the `external_transition` functions depend on a variable representing the elapsed time. The `internal_transition` functions also depend on time in a sense, as they are invoked when the time produced by the `time_advance` function elapses. Dependence on time turns out to be the key to the formalism’s flexibility, though that would not be obvious to a DEVS-unaware developer of object-oriented code.

One could of course use objects for models, keep models and simulators separate, and ensure that all transitions functions are informed of the elapsed time. In that case, one has essentially adopted DEVS, which is what we recommend. Remember that DEVS is only a set of conventions, and can be used in conjunction with various programming techniques.

7 Conclusion

We have demonstrated how DEVS impacts simulation from a developer's point of view. A particular building simulation was tackled with both a traditional approach to simulation development, and with DEVS. The simulation results were the same in both cases, but the DEVS-based code was organized in a way that we believe would enable and encourage collaboration. If adopted in practice, the DEVS-based approach may well improve matters from a simulation user's perspective as well, as collaborative development efforts would yield improved building performance simulation software.

References

- [1] Martina Pasini Livio Mazarella. Building Energy Simulation and Object-Oriented Modelling: Review and Reflections upon Achieved Results and Further Developments. In *Proceedings of the International IBPSA Conference (IBPSA)*, Glasgow, Scotland, 2009.
- [2] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2000.
- [3] Bernard P. Zeigler. Systems Movement: Autobiographical Retrospectives. *International Journal of General Systems*, 32(3):221–236, 2003.
- [4] Rhys Goldstein. *DEVS-Based Dynamic Simulation of Deformable Biological Structures*. PhD thesis, Carleton University, Ottawa, ON, Canada, 2009.
- [5] Gabriel A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, 2009.
- [6] Hans L. M. Vangheluwe. DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design*, Anchorage, AK, USA, 2000.