Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

AN INTRODUCTION TO MODULAR MODELING AND SIMULATION WITH PYTHONPDEVS AND THE BUILDING-BLOCK LIBRARY PYTHONPDEVS-BBL

Yentl Van Tendeloo
Randy Paredis
Hans Vangheluwe

Department of Mathematics and Computer Science – Flanders Make
University of Antwerp
Middelheimlaan 1
Antwerp, BELGIUM

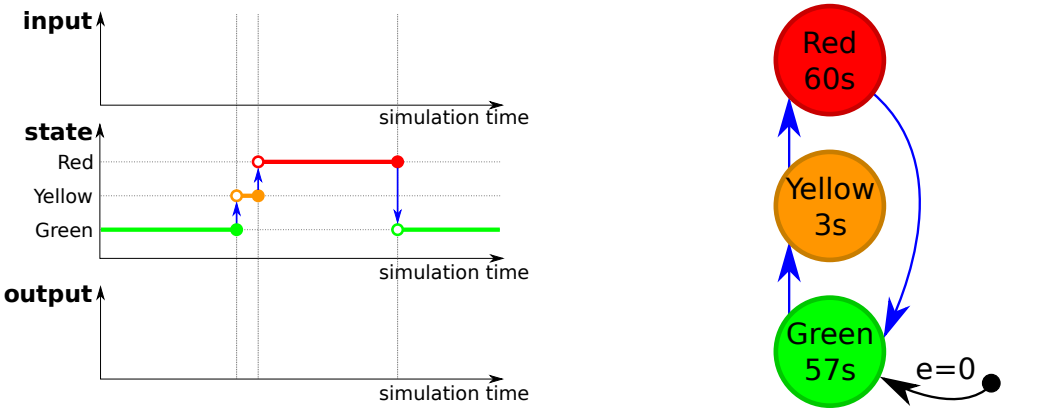
ABSTRACT

The Discrete Event System Specification (DEVS) is a popular formalism devised by Bernard Zeigler in the late 1970s for modeling complex dynamical systems using a discrete event abstraction. At this abstraction level, a timed sequence of pertinent "events" input to a system (or internal timeouts) causes instantaneous changes to the state of the system. Main advantages of DEVS are its precise, implementation independent specification, and its support for modular composition. This tutorial introduces the Classic DEVS formalism in a bottom-up fashion, using a simple traffic light example. The syntax and operational semantics of Atomic (*i.e.*, non-hierarchical) models are introduced first. Coupled (*i.e.*, hierarchical) models are introduced to structure and couple Atomic models. We continue to actual applications of DEVS, with an example in performance analysis of queueing systems. This uses generator, queue, etc. components from our PythonPDEVS Building Block Library. All examples in the paper are presented using the language PythonPDEVS and its simulator, though this introduction is equally applicable to other DEVS implementations. We conclude with further reading on DEVS theory, variants, and tools.

1 INTRODUCTION

The Discrete Event System Specification (DEVS) (Zeigler et al. 2000) first introduced by Bernard Zeigler in the late 1970s is a popular formalism for modeling complex dynamic systems using a discrete event abstraction. At this abstraction level, a timed sequence of pertinent "events" input to a system cause instantaneous changes to the state of the system. These events can be generated externally (*i.e.*, by another model, of the system's environment) or internally (*i.e.*, by the model itself due to timeouts). If a system has no (or does not react to) external events, it is called autonomous. The next state of the system is defined based on both the previous state of the system and the event. Between events, the state does not change, resulting in a piecewise constant state trajectory. Simulation kernels may thus only consider times at which events occur, skipping over all intermediate points in time. This is in contrast to discrete time models, where time is increased with a fixed increment, and only at those times the state is updated. Discrete event models have the advantage that their time granularity can, in theory, become arbitrarily small or large. However, only a finite number of events are allowed in any finite time span. Without this restriction, discrete event would become equivalent to continuous time. The added complexity of the discrete event abstraction makes it less appropriate for systems that naturally have a fixed time step.

This tutorial provides an introduction to DEVS (often referred to as Classic DEVS) using a simple traffic light example. This paper is a revised and extended version of our tutorial paper at the 2019 Winter Simulation Conference (Van Tendeloo, Vangheluwe, and Franceschini 2019). We start from a simple autonomous traffic light, incrementally extended up to a trafficlight with policeman interaction. Each



(a) Trace of the traffic light.

(b) Trace generating model.

Figure 1: Model of an autonomous traffic light.

increment introduces a new aspect of the DEVS formalism and the corresponding (informal) semantics. Each comes with an example implementation in PythonPDEVS (Van Tendeloo and Vangheluwe 2016), though the concepts are equally applicable to other tools. DEVS is a deterministic formalism rooted in automata theory. Simulating stochastic models becomes possible when sampling from distributions inside the time advance, output and state transition functions of a model. This is supported in PythonPDEVS-BBL, a Building Block Library consisting of re-usable model components. Note that sampling is based on pseudo-random number generators, which, though generating streams of numbers with desirable statistical properties, are deterministic once their seed is fixed. This supports repeatability of simulation experiments.

Atomic (*i.e.*, non-hierarchical) models are introduced in Section 2, coupled (*i.e.*, hierarchical) models in Section 3. Section 4 moves away from the traffic light example and presents a more complex queueing problem. A solution is presented based on the connection of components from the PythonPDEVS-BBL. Section 5 presents several directions of further reading on DEVS. Finally, Section 6 summarizes the tutorial.

2 ATOMIC DEVS MODELS

Atomic DEVS models are the indivisible building blocks of a model, describing system behavior. Throughout this section, we build up the complete formal specification of an atomic model, introducing new concepts as they are needed. In each intermediate step, we show and explain the concepts we introduce, how they are present in the running example, and how they influence the semantics. Additionally, example code illustrates the close match between the formal definition and the encoding in PythonPDEVS.

2.1 Autonomous Model

The simplest form of a traffic light is an autonomous traffic light, *i.e.*, without interaction with its environment. Observing the traffic light, we expect to see a trace similar to that in Figure 1a. Figure 1b gives an intuitive visual representation of a model that could generate such a trace. Trying to precisely describe Figure 1b, we distinguish these elements:

1. **Sequential State Set** S
   The basis of modeling traffic light behavior is the states (colors) it can be in. These states are *sequential*: the traffic light can only be in one of these states at a given time. The dynamics of the system consists of transitions between these states. The term sequential stems from automata theory. In contrast to automata, the DEVS state set may however be infinite (*e.g.*, $\mathbb{R}$).
2. **Time Advance Function** $ta : S \to \mathbb{R}^{+}_{0,+\infty}$

The system stays in each state $s \in S$ for a certain duration of time before spontaneously making a transition to the next state. This duration is modelled using the time advance function ta, defined for each and every element of the state set. The duration can be any positive real number, including zero and infinity. A negative time is not allowed, as this would model time progressing backwards. DEVS allows a time advance of 0, as an abstraction of fast real-world dynamics. In this case, a very small delay might be irrelevant to properties of interest of the system being modeled, and be replaced by 0 without affecting the validity of the model. A state can also be an artificial transient state without any real-world equivalent. The latter is used to overcome constraints imposed by the formalism, as discussed in section 2.3. It should be used with caution. DEVS does not consider time units, despite the use of seconds in our visualization. Simulation time is just a real number, and the interpretation given to it is up to the user. The time e spent in a state since last entering it is called the *elapsed time*. For any $s \in S$, e evolves from 0, when entering state s, to $ta(s)$, when leaving that state.

3. **Total State Set** Q
   Only the sequential state $s \in S$ augmented with the elapsed time captures the state a system is in entirely. This leads to the notion of total state q, an element of the total state set $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$.
4. **Internal Transition Function** $\delta_{int} : S \to S$
   The internal transition function δ_{int} specifies which next state to transition to from a given state s once the system has been in that state for $ta(s)$. If $ta(s) = 0$, state s is called *transient*. If $ta(s) = +\infty$, the system is called *passivated* in state s. The internal transition function value for that state is irrelevant (and may be omitted in some implementations such as PythonPDEVS) as the system is stuck in that state and δ_{int} will never be called. As it is a function, every state has at most one next state, keeping the specification deterministic. The function does not have to be injective: some states have the same state as next state.
5. **Initial Total State** $q_{init} \in Q$
   We also need to specify the initial state $q_{init} = (s_{init}, e_{init})$ with $s_{init} \in S$ and $e_{init} \in [0, ta(s_{init})]$. To the simulator, it will seem as if the model has already been in the initial state s_{init} for e_{init} at the start of a simulation. While this is not present in the original specification of the DEVS formalism by Zeigler et al. (2000), it is essential to unequivocally specify a re-usable model.

Note how only the internal transition function is described as changing the state. Therefore, the other functions such as the time advance do not modify the state. From an implementation point of view, their state access is read-only.

We describe the model in Figure 1b as the 4-tuple defined in Figure 2a. Figure 2b presents the example specification as PythonPDEVS code.

Algorithm 1 Simulation pseudo-code for autonomous models.

```
time ← 0
current_state ← initial_state
last_time ← −initial_elapsed
while not termination_condition(current_state, time) do
    time ← last_time + ta(current_state)
    current_state ← δ_int(current_state)
    last_time ← time
end while
```

For this simple formalism, we define the semantics as in Algorithm 1. The model is initialized with simulation time set to 0, and the state set to the initial state (*i.e.*, GREEN). Simulation updates the time with

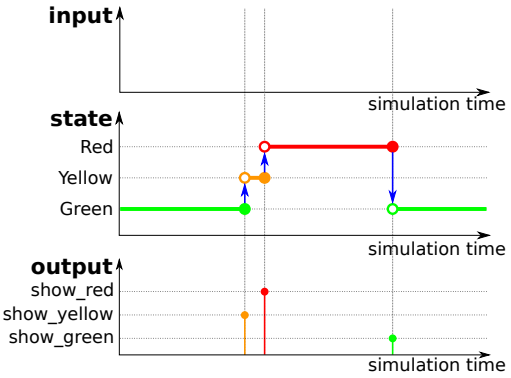
$$\langle S, q_{init}, \delta_{int}, ta \rangle$$
$$S = \{\text{GREEN}, \text{YELLOW}, \text{RED}\}$$
$$q_{init} = (\text{GREEN}, 0.0)$$
$$\delta_{int} = \{\text{GREEN} \rightarrow \text{YELLOW},$$
$$\text{YELLOW} \rightarrow \text{RED},$$
$$\text{RED} \rightarrow \text{GREEN}\}$$
$$ta = \{\text{GREEN} \rightarrow 57,$$
$$\text{YELLOW} \rightarrow 3,$$
$$\text{RED} \rightarrow 60\}$$

(a) Atomic DEVS model.

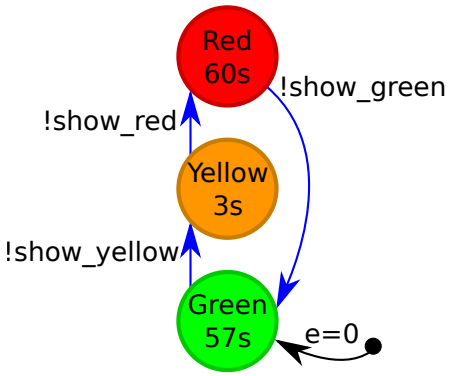
```
from pypdevs.DEVS import *

class TrafficLightAutonomous(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self, "Light")
    self.state, self.elapsed = ("Green", 0.0)

  def intTransition(self):
    state = self.state
    return {"Red": "Green",
            "Yellow": "Red",
            "Green": "Yellow"}[state]

  def timeAdvance(self):
    state = self.state
    return {"Red": 60,
            "Yellow": 3,
            "Green": 57}[state]
```

(b) PythonPDEVS representation of Figure 2a.

Figure 2: Atomic DEVS model of the autonomous traffic light.



(a) Trace of the traffic light.



(b) Trace generating model.

Figure 3: Model of the autonomous traffic light with output.

the returnvalue of the time advance function, and executes the internal transition function on the current state to get the new state. This is repeated until simulation terminates.

2.2 Autonomous Model With Output

Recall that DEVS is a modular formalism, with only the atomic model having access to its internal state. This raises a problem for our traffic light: others have no access to the current state (*i.e.*, its color).

We, therefore, want the traffic light to output an event indicative of its current color, in this case in the form of a string (and not as the element of an enumeration). For now, the output is tightly linked to the set of state, but this will not remain the case. Our desired trace is shown in Figure 3a. We see that we now output events indicating the start of the specified period. Recall, also, that DEVS is a discrete event formalism: the output is only a single event at some discrete point in time and is not a continuous signal. The receiver of the event thus would have to store the event to know the current state of the traffic light at any given point in time. Visually, the model is updated to Figure 3b, using the exclamation mark on a transition to indicate output generation. Output only happens at an internal transition.

$$\langle Y, S, q_{init}, \delta_{int}, \lambda, ta \rangle$$

$$Y = \{show\_green, show\_yellow, show\_red\}$$

$$S = \{\text{GREEN}, \text{YELLOW}, \text{RED}\}$$

$$q_{init} = (\text{GREEN}, 0.0)$$

$$\delta_{int} = \{\text{GREEN} \rightarrow \text{YELLOW}, \text{YELLOW} \rightarrow \text{RED}, \text{RED} \rightarrow \text{GREEN}\}$$

$$\lambda = \{\text{GREEN} \rightarrow show\_yellow, \text{YELLOW} \rightarrow show\_red, \text{RED} \rightarrow show\_green\}$$

$$ta = \{\text{GREEN} \rightarrow 57, \text{YELLOW} \rightarrow 3, \text{RED} \rightarrow 60\}$$

(a) Atomic DEVS model.

```
from pypdevs.DEVS import *

class TrafficLightWithOutput(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self, "Light")
    self.state, self.elapsed = ("Green", 0.0)
    self.observe = self.addOutPort("observer")

  def intTransition(self):
    state = self.state
    return {"Red": "Green",
            "Yellow": "Red",
            "Green": "Yellow"}[state]

  def timeAdvance(self):
    state = self.state
    return {"Red": 60,
            "Yellow": 3,
            "Green": 57}[state]

  def outputFnc(self):
    state = self.state
    out_map = {"Red": "show_green",
               "Yellow": "show_red",
               "Green": "show_yellow"}
    return {self.observe: out_map[state]}
```

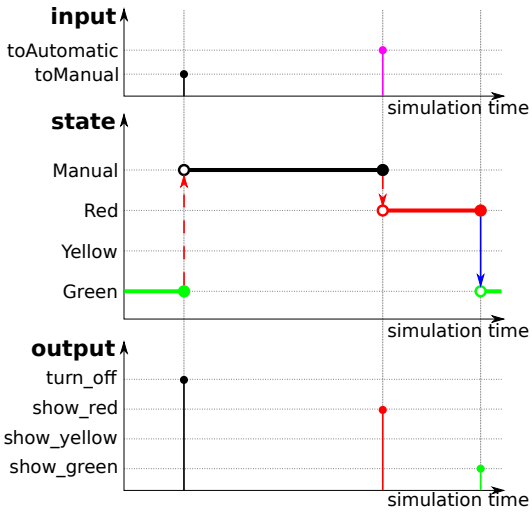
(b) PythonPDEVS representation of Figure 4a.

Figure 4: Atomic DEVS model of the autonomous traffic light with output.

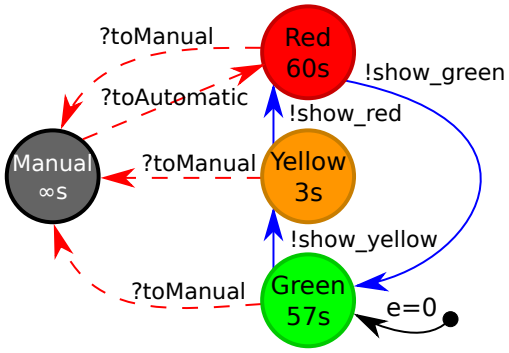
Analysing the updated model, we see that two more concepts are required to allow for output.

1. **Output Events Set** Y
   Similar to the set of states, we should also define the set of output events. This set serves as an interface to other components, defining the events it may produce as output. Events can have a complex internal structure, which is however mathematically equivalent to a flat event set. If l output ports are used, each port has its own output set Y_i and $Y = \times_{i=1}^{l} Y_i$.
2. **Output Function** $\lambda : S \rightarrow Y \cup \{\phi\}$
   With the set of output events defined, we still need to generate the events. Similar to the other functions, the output function is defined on the state, and deterministically returns an event (or the *null* event ϕ denoting the absence of output). As seen in Figure 3b, the event is generated *before* the new state is reached. This means that the output function still uses the old state (*i.e.*, the one that is being left) instead of the new state. For this reason, the output function needs to be invoked right before the internal transition function. In the case of our traffic light, the output function needs to return the color of the *next* state, instead of the current state. For example, if the output function receives the GREEN state as input, it needs to generate a *show_yellow* event.
   Similar to the time advance function, this function does not output a new state, and therefore state access is read-only. This might require some workarounds: outputting an event often has some repercussions on the model state, such as removing the event from a queue. Since the state can not be written to, these changes need to be remembered and executed as soon as the internal transition is executed.

We describe the model in Figure 3b as the 6-tuple defined in Figure 4a. Figure 4b presents the example specification as PythonPDEVS code.



(a) Trace of the autonomous traffic light.



(b) Naive model that should generate the trace in Figure 5a.

Figure 5: Trace and naive model of the interruptable traffic light.

The pseudo-code is slightly altered to include output generation, as shown in Algorithm 2. Recall that output is generated before the internal transition is executed, so the method invocation happens right before the transition.

Algorithm 2 DEVS simulation pseudo-code for autonomous models with output.

```
time ← 0
current_state ← initial_state
last_time ← −initial_elapsed
while not termination_condition(current_state, time) do
    time ← last_time + ta(current_state)
    output(λ(current_state))
    current_state ← δ_int(current_state)
    last_time ← time
end while
```

2.3 Interruptable Model

Our current traffic light specification is still completely autonomous. While this is fine in most circumstances, a policeman might want to temporarily shut down the traffic lights when they wish to manage traffic manually. To allow for this, our traffic light must process external events: the event from the policeman to shutdown and to start up again. Figure 5a shows the trace we wish to obtain. A model generating this trace is shown in Figure 5b, using a question mark to indicate event reception.

We once more require two additional elements in the DEVS specification.

1. **Input Events Set** X
   Similar to the output events set, we need to define the events we expect to receive. This is again a definition of the interface, such that other components know which events are understood by this

model. Events can have a complex internal structure, which is however mathematically equivalent to a flat event set. If m input ports are used, each port has its own input set X_i and $X = \times_{i=1}^{m} X_i$.

2. **External Transition Function** $\delta_{ext} : Q \times X \to S$
   Similar to the internal transition function, the external transition function assigns a new state. The external transition function is still dependent on the current state, just like the internal transition function. The external transition function has access to two more values: the elapsed time (*i.e.*, it depends on the total state), and the input event that trigerred it. The *elapsed time* is the time since the last transition (either internal or external). Whereas this number was implicitly known in the internal transition function (*i.e.*, the value of the time advance function), here it needs to be passed explicitly. Elapsed time is a number in the range $[0, ta(s)]$, with s the current state of the model. Both 0 and $ta(s)$ are included in the range: it is possible to receive an event right after a transition happened, or right before an internal transition happens. The received event is the final parameter to this function.

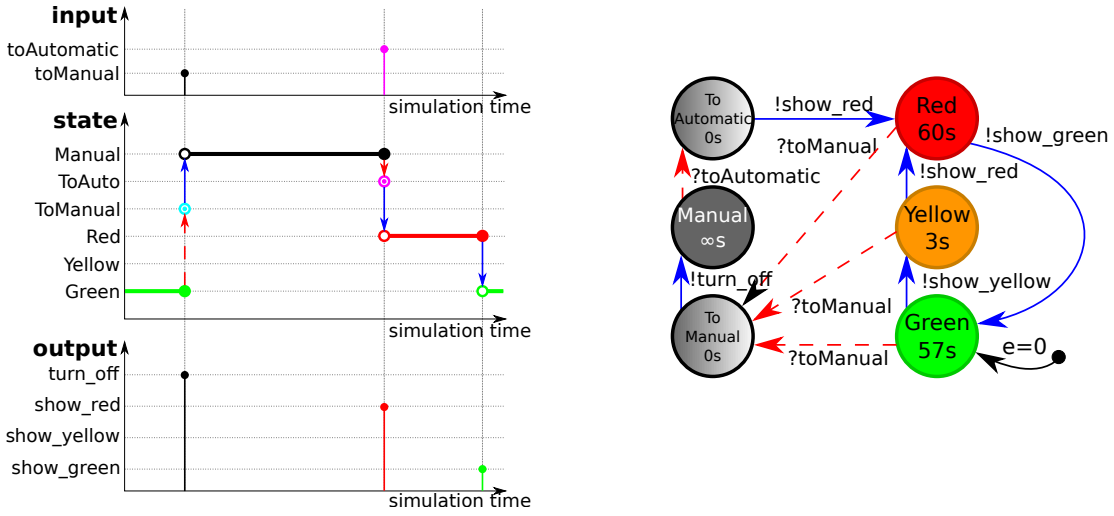
Summarizing the above, an atomic DEVS model can be defined as an 8-tuple: $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$.

Algorithm 3 presents the complete semantics of an atomic model in pseudo-code. Similar to before, we still have the same simulation loop, but now we can be interrupted externally. At each time step, we need to determine whether an external interrupt occurs before the internal interrupt is scheduled. If that is not the case, we simply continue like before, by executing the internal transition. Note that no output is generated with an external transition. Also, if an internal and external transition occur at the same time, only the interrupting external transition will be taken.

Algorithm 3 DEVS simulation pseudo-code for interruptable models.

```
time ← 0
current_state ← initial_state
last_time ← −initial_elapsed
while not termination_condition(current_state, time) do
  next_time ← last_time + ta(current_state)
  if time_next_event ≤ next_time then
    elapsed ← time_next_event − last_time
    current_state ← δ_ext((current_state, elapsed), next_event)
    time ← time_next_event
  else
    time ← next_time
    output(λ(current_state))
    current_state ← δ_int(current_state)
  end if
  last_time ← time
end while
```

While we now have all elements of the DEVS specification for atomic models, we are not done yet with out traffic light model. When we include the additional state MANUAL, we also need to send out an output message indicating that the traffic light is off. But, recall that an output function was only invoked before an internal transition, not before an external transition. To nonetheless have an output, we need to make sure that an internal transition happens before we actually reach the MANUAL state. This can be done through the introduction of an artificial intermediate state, which times out immediately, and sends out the *turn_off* event. Instead of going to MANUAL upon reception of the *toManual* event, we go to the artificial state TOMANUAL. The time advance of this state is set to 0, since it is only an artificial state



(a) Trace of model with corrected artificial states.

(b) Trace generating model.

Figure 6: Trace and corrected model of the interruptable traffic light.

without any meaning in the domain under study. Its output function is triggered immediately after, due to the time advance of zero, and the *turn_off* output is generated while transferring to MANUAL. Similarly, when we receive the *toAutomatic* event, we need to go to an artificial TOAUTOMATIC state to generate the *show_red* event. A visualization of the corrected trace and corresponding model is shown in Figure 6a and Figure 6b respectively.

We describe the model in Figure 6b as the 8-tuple defined in Figure 7a. Figure 7b presents the example specification as PythonPDEVS code.

3 COUPLED DEVS MODELS

While our traffic light example is able to receive and output events, there are no other (atomic) models to communicate with. Atomic models can be connected in a network of communicating components in the form of coupled models. This is done in the context of our previous traffic light model, which will be connected to a policeman model. The details of the traffic light are exactly as before; the details of the policeman are irrelevant here, as long as it outputs *toAutomatic* and *toManual* events.

3.1 Basic Coupling

The first problem we encounter with coupling the traffic light and policeman is the structure: how do we define a set of models and their interactions? This is the core definition of a coupled model: it is merely a structural model that couples models. Contrary to the atomic models, there is *no behavior* associated with a coupled model. Behavior is the responsibility of atomic models, and structure that of coupled models.

To define the basic structure, we need three elements.

1. **Submodel Index Set** D
   The set D of model instance references (descriptor) specifies which models make up this coupled model. Note that we use $self$ to refer to the coupled model itself.
2. **Submodel Specifications** $\{M_i = \langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle | i \in D\}$
   For each submodel index in D, we include the 8-tuple describing the atomic model of the model being referred to. Here, all submodels of a coupled DEVS model are atomic models. As mentioned

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

$$X = \{toAutomatic, toManual\}$$

$$Y = \{show\_green, show\_yellow, show\_red, turn\_off\}$$

$$S = \{\text{GREEN}, \text{YELLOW}, \text{RED}, \text{TOMANUAL}, \text{TOAUTOMATIC}, \text{MANUAL}\}$$

$$q_{init} = (\text{GREEN}, 0.0)$$

$$\delta_{int} = \{\text{GREEN} \to \text{YELLOW}, \text{YELLOW} \to \text{RED}, \text{RED} \to \text{GREEN}, \text{TOMANUAL} \to \text{MANUAL}, \text{TOAUTOMATIC} \to \text{RED}\}$$

$$\delta_{ext} = \{((\text{GREEN}, *), toManual) \to \text{TOMANUAL}, ((\text{YELLOW}, *), toManual) \to \text{TOMANUAL}, ((\text{RED}, *), toManual) \to \text{TOMANUAL}, ((\text{MANUAL}, *), toAutomatic) \to \text{TOAUTOMATIC}\}$$

$$\lambda = \{\text{GREEN} \to show\_yellow, \text{YELLOW} \to show\_red, \text{RED} \to show\_green, \text{TOMANUAL} \to turn\_off, \text{TOAUTOMATIC} \to show\_red\}$$

$$ta = \{\text{GREEN} \to 57, \text{YELLOW} \to 3, \text{RED} \to 60, \text{MANUAL} \to +\infty, \text{TOMANUAL} \to 0, \text{TOAUTOMATIC} \to 0\}$$

(a) Interruptable DEVS model.

```
from pypdevs.DEVS import *
from pypdevs.infinity import INFINITY

class TrafficLight(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self, "Light")
    self.state = "Green"
    self.elapsed = 0.0
    self.observe = \
        self.addOutPort("observer")
    self.interrupt = \
        self.addInPort("interrupt")

  def intTransition(self):
    state = self.state
    return {"Red": "Green",
            "Yellow": "Red",
            "ToManual": "Manual",
            "ToAutomatic": "Red",
            "Green": "Yellow"}[state]

  def timeAdvance(self):
    state = self.state
    return {"Red": 60,
            "Yellow": 3,
            "Green": 57,
            "ToManual": 0,
            "ToAutomatic": 0,
            "Manual": INFINITY}[state]

  def outputFnc(self):
    state = self.state
    out_map = {"Red": "show_green",
               "Yellow": "show_red",
               "ToManual": "turn_off",
               "ToAutomatic": "show_red",
               "Green": "show_yellow"}
    return {self.observe: out_map[state]}

  def extTransition(self, inputs):
    inp = inputs[self.interrupt]
    if inp == "toManual":
      return "ToManual"
    elif inp == "toAutomatic":
      if self.state == "Manual":
        return "ToAutomatic"
```

(b) PythonPDEVS representation of Figure 7a.

Figure 7: Abstract and concrete syntax of the full traffic light.

in Section 5.1, any coupled model can be "flattened" to an equivalent atomic model. As such coupled models may in general be composed of coupled submodels.

3. **Model Influencees** $\{I_i \in 2^{D \cup \{self\}} | i \in D\}$
   Apart from defining the model instances and their specifications, we need to define the connections between them. Connections are defined through the use of influencee sets: for each atomic model instance, we define the set of models influenced by that model. There are some limitations on couplings, to disallow inconsistent models.
   First, a model should not influence itself ($\forall i \in D \cup \{self\} : i \notin I_i$). While there is no significant problem with this in itself, it would cause the model to trigger both its internal and external transition simultaneously.

Second, only links within the coupled model and between its input and output ports and the submodels are allowed: $\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$. This is another way of saying that connections should respect modularity.
There is no explicit constraint on algebraic loops (*i.e.*, a self loop of connected models that all have a time advance 0, preventing the progression of simulated time). If this situation is not resolved, it is possible for simulation to get stuck at that specific point in time. In that case, the model is called *illigitimate*. Simulator implementations often introduce a number-op-zero-time-advance iteration counter to detect this, at simulation time.

A coupled model can thus be defined as a 3-tuple: $\langle D, \{M_i\}, \{I_i\} \rangle$.

3.2 Input and Output

Our coupled model now combines two atomic models. While it is now possible for the policeman to pass the event to the traffic light, we again lost the ability to send out the state of the traffic light. The events can't reach outside of the current coupled model. Therefore, we augment the coupled model with **input and output events** (X_{self} and Y_{self}, respectively), serving as the interface to the coupled model. A coupled model can thus be defined as a 5-tuple: $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\} \rangle$.

The constraints on the couplings need to be relaxed to accomodate for the new capabilities of the coupled model: a model can be influenced by the input events of the coupled model, and likewise the models can also influence the output events of the coupled model. The previously defined constraints over the influencees are relaxed from D to $D \cup \{self\}$, to allow for connections to and from the coupled model.

3.3 Tie-breaking

Recall that DEVS is considered a formal and precise formalism. But, while all components are precisely defined, their interplay is not completely defined yet: what happens when the traffic light changes its state at exactly the same time as the policeman performs its transition? Would the traffic light switch on to the next state first and then process the policeman's interrupt, or would it directly respond to the interrupt, ignoring the internal event? While it is a minimal difference in this case, the state reached after the timeout might respond significantly different to the incoming event.

DEVS solves this problem by defining a **tie-breaking function** ($select : 2^D \rightarrow D$). This function takes all conflicting models and returns the one that gets priority over the others. After the execution of that internal transition, and possibly the external transitions that it caused elsewhere, it might be that the set of imminent models has changed. If multiple models are still imminent, we repeat the above procedure (potentially invoking the *select* function again with the new set of imminent models).

This new addition changes the coupled model to a 6-tuple: $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, select \rangle$.

3.4 Translation Functions

Finally, in this case we had full control over both atomic models that are combined. We might not always be that lucky, as it is possible to reuse atomic models defined elsewhere. Depending on the application domain of the reused models, they might work with different events. For example, if our policeman and traffic light were both predefined, with the policeman using *go_to_work* and *take_break* and the traffic light listening to *toAutomatic* and *toManual*, it would be impossible to directly couple them. While it is possible to define wrapper blocks (*i.e.*, artificial atomic models that take an event as input and, with time advance zero, output the translated version), DEVS provides a more elegant solution to this problem.

Connections are augmented with a **translation function** ($Z_{i,j}$), specifying how the event that enters the connection is translated before it is handed over to the endpoint of the connection. The function thus maps output events to input events, potentially modifying their content.

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

$$X_{self} = \{\}$$
$$Y_{self} = \{\}$$
$$D = \{light, police\}$$
$$M_{light} = \langle ... \rangle$$
$$M_{police} = \langle ... \rangle$$
$$I_{light} = \{\}$$
$$I_{police} = \{light\}$$
$$\forall i, j \in \{police, light, self\} : Z_{i,j} = id$$
$$select = \{\{police, light\} \rightarrow police,$$
$$\{police\} \rightarrow police,$$
$$\{light\} \rightarrow light\}$$

(a) Coupled DEVS model.

```
from pypdevs.DEVS import *

from trafficlight import TrafficLight
from policeman import Policeman

class TrafficLightSystem(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self, "system")
    self.light = \
        self.addSubModel(TrafficLight())
    self.police = \
        self.addSubModel(Policeman())
    self.connectPorts(self.police.out,
                      self.light.interrupt)

  def select(self, imm):
    if self.police in imm:
        return self.police
    else:
        return self.light
```

(b) PythonPDEVS representation of Figure 8a.

Figure 8: Coupled DEVS model and its PythonPDEVS representation of the system. Atomic DEVS models not shown for brevity.

$$\begin{array}{lll} Z_{self,j} & : X_{self} \rightarrow X_j & \forall j \in D \\ Z_{i,self} & : Y_i \rightarrow Y_{self} & \forall i \in D \\ Z_{i,j} & : Y_i \rightarrow X_j & \forall i, j \in D \end{array}$$

These translation functions are defined for each connection, including those between the coupled model's input and output events: $\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$. The translation function is implicitly assumed to be the identity function if it is not defined. In case an event needs to traverse multiple connections, all translation functions are chained in order of traversal.

With the addition of this final element, we a coupled model takes the form of a 7-tuple:

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

Figure 8a gives the full description of our traffic light – policeman model. Figure 8b presents the example specification as PythonPDEVS code. In PythonPDEVS, the translation function is an optional third parameter of the `connectPorts` method. By default, the identity function is used.

For a precise description of coupled model semantics, we refer to Section 5 for further reading.

4 APPLICATION TO QUEUEING SYSTEMS

DEVS is highly suited to model systems whose behavior involves competition for shared resources leading to queueing. We briefly present a simple queueing problem and describe results obtained through DEVS modeling and simulation. Due to space restrictions, the used models and an elaborate explanation of their specification can be found at https://msdl.uantwerpen.be/documentation/PythonPDEVS/queueing.html.

4.1 Problem Description and Building Block Library

In this example, we model arriving people who enter a simple queue that gets served by multiple processors. Instances of this type of queueing systems are widespread, such as, for example, in airport security. Our model is parameterizable in several ways: we can define the distributions used for event generation: person inter-arrival times and person "size" (how long it will take to process that person), the number of processors, performance (service time) of each individual processor, and the scheduling policy of the queue when

```
Generators                  Queues                  Routing             Mathematics         I/O
    Constant Generator          Simple Queue            Finish              Adder               Logger
    Function Generator          Queue Tracker           Halt                Multiplier          Emergency
    Table Generator             Queue                   Choose Input        Equation            Alert
    Single Fire                 Retain                  Choose Output       Differentiator      Critical
    Bulk Generator              Advance                 Pick                Integrator          Error
    Random Number Generator                             Guard               Random              Warn
    Random Delay Generator  Transformers                Gate                                    Notice
                                Transformer             Delayer                                 Info
Collectors                      Lookup Table            Timer                                   Debug
    Collector                   Pack                    Sync                                    File Writer
    Estimate Collector          Unpack                                                          File Reader
    Table Collector                                                                             Listener
    Counter
```

Figure 9: PythonPDEVS Building Block Library (BBL)

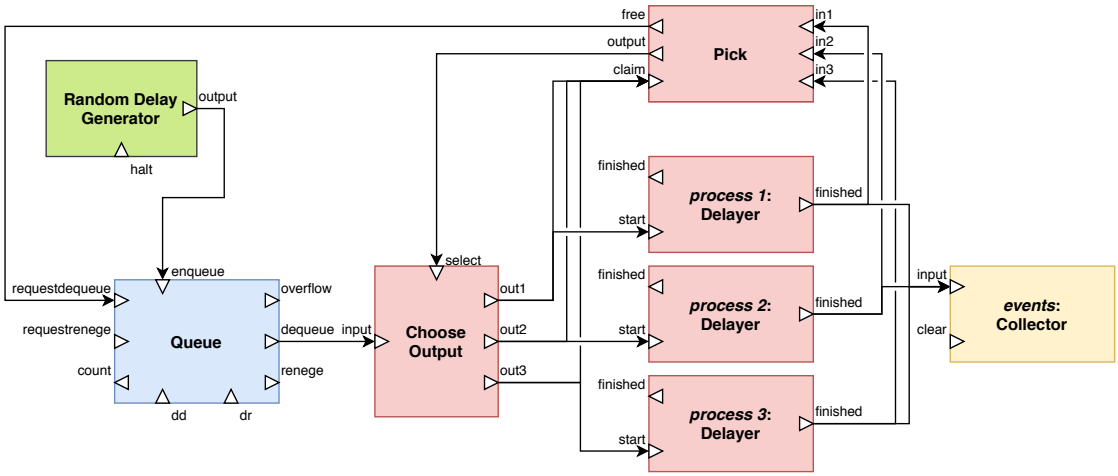


Figure 10: Example model using PythonPDEVS Building Block Library components

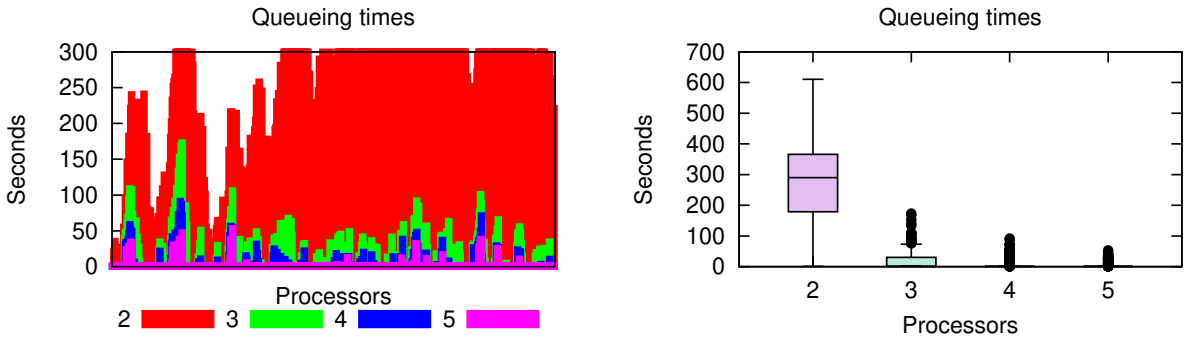
selecting a processor. For our performance analysis, we are interested in the influence of the number of processors (*e.g.*, metal detectors) on the average and maximal queueing time of jobs (*e.g.*, travellers).

Events (person arrivals) are produced by a generator based on an Inter-Arrival Time distribution. They enter the queue, which decides the processor that they will be sent to. The queue works First-In-First-Out (FIFO) in case multiple events are queueing. For a processor to signal that it is available, it needs to notify the queue. The queue keeps track of available processors. When an event arrives at a processor, it is processed for some time determined by the size of the event and the performance characteristics of the processor. After processing, the processor signals the queue and sends out the event that was being processed.

As the components of the example model are quite common, a library of re-usable components is very useful. Figure 9 lists the blocks available in PythonPDEVS-BBL, a Building Block Library for PythonPDEVS. The block names are self-explanatory. Their detailed and precise semantics is given by their DEVS specification. Similar block libraries are found in other discrete event modeling tools. Modeling the example problem is now reduced to connecting blocks from the library as shown in Figure 10.

4.2 Performance Analysis

We now run the simulation experiment with the models provided online at https://msdl.uantwerpen.be/documentation/PythonPDEVS/queueing.html. This results in a trace file in CSV format from which relevant performance measures such as average queueing times, maximal queueing times, number of events, processor utilization, and so on are obtained.



(a) Evolution of queueing times for subsequent customers.

(b) Boxplot of queueing times for varying number of active processors.

Figure 11: Analysis of the queueing system from Figure 10.

Our initial goal was to find out the influence of the number of processors on the average and maximum queueing time. Figure 11a shows the evolution of the waiting time for successive clients. Figure 11b shows the same results, drawn using boxplots. These results indicate that while two processors are able to handle the load, the maximum waiting time is too high: a median of 300 seconds and a maximum of around 600 seconds. When a single additional processor is added, average waiting time decreases significantly, and the maximum waiting time becomes acceptable: an average job is served immediately, with 75% of jobs handled within 25 seconds. Further addition of processors has a positive effect on queueing times, but this might not warrant the increased cost: apart from some exceptions, all customers are processed immediately starting from four processors. Ideally, a cost function would be defined to quantify the value (or dissatisfaction) of waiting jobs, and compare this to the cost of adding additional processors. We can, then, optimize that cost function to find out the ideal balance between paying more for additional processors and losing money due to long job processing times.

5 FURTHER READING

We consider three main directions of interest: DEVS theory, DEVS variants, and DEVS tools.

5.1 Theory

Our bottom-up introduction to DEVS has been example-driven. This forced us to drop some theoretical concepts, which can remain hidden to DEVS users. In particular, we did not go into the exact semantics of a coupled DEVS model, apart from an intuitive explanation of its meaning as a hierarchical model. This brings us to the closure under coupling property of DEVS, which states that any coupled model can be translated to a behaviorally equivalent atomic model. As the semantics of an atomic model are known, this effectively provides denotational semantics for coupled models. The description of closure under coupling, also called "flattening", can be found in (Zeigler et al. 2000). An efficient algorithm for symbolic flattening of models is presented by (Chen and Vangheluwe 2010). Another way of defining the semantics is the operational approach: providing similar pseudo-code for a coupled DEVS simulator, as we did for atomic models. This is called the *abstract simulator*, and can also be found in the original specification. These algorithms are not focused on performance. A performance-oriented view can be found in (Nutaro 2010). Additionally, an example-driven introduction to DEVS and its abstract simulator is given in (Wainer 2009). While we presented DEVS in the context of queueing systems simulation and performance analysis, it can also be used for purposes such as real-time application synthesis. It has been shown that DEVS can serve as a simulation assembly language, onto which other languages can be mapped (Vangheluwe 2000).

5.2 Variants

Many variants of DEVS address specific issues. Parallel DEVS is probably the most popular variant, and is often considered to be a replacement for the Classic DEVS formalism. It is similar to Classic DEVS but it allows for internal transitions to happen in parallel, thus removing the need for the (admittedly artificial) *select* function. This requires the addition of *bags* of events and a new *confluent transition function*. A description of Parallel DEVS can be found in (Chow and Zeigler 1994). Dynamic Structure DEVS (DSDEVS) allows to explicitly model dynamic structural changes, such as adding or removing new atomic or coupled models, and adding or removing connections between sub-models. While this can also be modeled in DEVS by manually expanding the set of allowed configurations, DSDEVS allows for simulator support, significantly increasing performance and ease-of-use. A description of DSDEVS can be found in (Barros 1995), and the abstract simulator in (Barros 1998). Variants of DSDEVS have also been created for Parallel DEVS (Barros 1997), and with different ways of representing the dynamicity (Uhrmacher 2001). Cell-DEVS is another variant of the DEVS formalism, which merges Cellular Automata with DEVS. Model specification is similar to Cellular Automata models, but the underlying formalism used for simulation is DEVS. This allows the continuous time base of DEVS to be used for Cellular Automata models. Many other variants exist, each with their own focus.

5.3 Tools

Our introduction to DEVS has made use of PythonPDEVS (Van Tendeloo and Vangheluwe 2016), which is an efficiency-oriented (Van Tendeloo and Vangheluwe 2014) and distributed (Van Tendeloo and Vangheluwe 2015) DEVS simulator. Nonetheless, all concepts introduced in this tutorial are applicable to other DEVS simulation tools as well. ADEVS (Nutaro 2015) is a minimalistic, highly efficient, C++ implementation of the Parallel DEVS formalism. A recent C++-based addition is Cadmium (Belloli et al. 2019), a C++17 header-only Parallel DEVS simulator. DEVS-Suite (Kim et al. 2009) is a full modeling and simulation environment implemented in Java, with a visual simulation interface. DEVSimPy (Capocchi et al. 2011) is a modeling and simulation tool which relies on PythonPDEVS as its simulation kernel. A Parallel DEVS debugging extension (Van Mierlo et al. 2017) allows for fine-grained debugging, based on the PythonPDEVS simulation kernel. DesignDEVS (Goldstein et al. 2016) is an intuitive DEVS simulator. DEVS-Ruby (Franceschini et al. 2014) is a DEVS simulator written in Ruby. Several comparisons exist between different tools, such as a comparison of their interface, features and performance (Van Tendeloo and Vangheluwe 2017), and an in-depth comparison of performance (Risco-Martín et al. 2017).

6 SUMMARY

This tutorial briefly presented the core concepts of DEVS, a popular formalism for the modeling of complex dynamic systems using a discrete event abstraction. An important use of DEVS is for the simulation of queueing networks, of which an example was given, using PythonPDEVS-BBL, a library of re-usable model components. Further reading on DEVS was provided with additional details on the theoretical aspects, a list of variants, and supporting tools.

ACKNOWLEDGEMENTS

This work was partly funded by a PhD fellowship grant from the Research Foundation – Flanders, and partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

Barros, F. J. 1995. “Dynamic Structure Discrete Event System Specification: a New Formalism for Dynamic Structure Modeling and Simulation”. In *Proceedings of the 1995 Winter Simulation Conference*, edited by C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, 781–785. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Barros, F. J. 1997. “Modeling Formalisms for Dynamic Structure Systems”. *ACM TOMACS* 7:501–515.

Barros, F. J. 1998. "Abstract Simulators for the DSDE Formalism". In *Proceedings of the 1998 Winter simulation Conference*, edited by D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, 407–412. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Belloli, L., D. Vicino, C. R. Martin, and G. A. Wainer. 2019. "Building DEVS Models with the Cadmium Tool". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 45–59. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Capocchi, L., J. F. Santucci, B. Poggi, and C. Nicolai. 2011. "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems". In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 170–175. Paris, France.

Chen, B., and H. Vangheluwe. 2010. "Symbolic Flattening of DEVS models". In *Proceedings of the 2010 Summer Simulation Multiconference*, 209–218. Ottawa, ON, Canada.

Chow, A. C. H., and B. P. Zeigler. 1994. "Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, 716–722. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Franceschini, R., P.-A. Bisgambiglia, P. Bisgambiglia, and D. Hill. 2014. "DEVS-Ruby: A Domain Specific Language for DEVS Modeling and Simulation (WIP)". In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS*, 103–108. Tampa, FL, USA.

Goldstein, R., S. Breslav, and A. Khan. 2016. "DesignDEVS: Reinforcing Theoretical Principles in a Practical and Lightweight Simulation Environment". In *Proceedings of the 2016 Spring Simulation Multiconference*, 2:1–2:8. Pasadena, CA, USA.

Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. 2009. "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring". In *Proceedings of the 2009 Spring Simulation Multiconference*, 161:1–161:7.

Nutaro, J. J. 2010. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. 1st ed. Wiley.

Nutaro, James J. 2015. "adevs". http://www.ornl.gov/~1qn/adevs/. Accessed 10 May 2020.

Risco-Martín, J. L., S. Mittal, J. C. Fabero Jiménez, M. Zapater, and R. Hermida Correa. 2017. "Reconsidering the Performance of DEVS Modeling and Simulation Environment Using the DEVStone Benchmark". *SIMULATION*.

Uhrmacher, A. M. 2001. "Dynamic Structures in Modeling and Simulation: a Reflective Approach". *ACM Transactions on Modeling and Computer Simulation* 11:206–232.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2017. "Debugging Parallel DEVS". *SIMULATION* 93(4):285–306.

Van Tendeloo, Y., and H. Vangheluwe. 2014. "The Modular Architecture of the Python(P)DEVS Simulation Kernel". In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS*, 97–102.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: a Distributed Parallel DEVS simulator". In *Proceedings of the 2015 Spring Simulation Multiconference*, 844–851.

Van Tendeloo, Y., and H. Vangheluwe. 2016. "An Overview of PythonPDEVS". In *JDF 2016*, 59–66: Cépaduès.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "An Evaluation of DEVS Simulation Tools". *SIMULATION* 93(2):103–121.

Van Tendeloo, Y., H. Vangheluwe, and R. Franceschini. 2019. "An Introduction to Modelling and Simulation with (Python(P))DEVS". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 1415 – 1429. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Vangheluwe, H. 2000. "DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling". In *IEEE International Symposium on Computer-Aided Control System Design*, 129–134. Anchorage, AK, USA.

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. 1st ed. CRC Press.

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

AUTHOR BIOGRAPHIES

YENTL VAN TENDELOO holds a PhD (2018) from the University of Antwerp (Belgium), in the Modelling, Simulation and Design Lab (MSDL). He developed the PythonPDEVS simulator in his 2014 Masters thesis. His e-mail address is Yentl.VanTendeloo@uantwerpen.be.

RANDY PAREDIS developed PythonPDEVS-BBL as part of his Masters thesis in the MSDL where he is currently a PhD student. His e-mail address is Randy.Paredis@student.uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium). where he heads the MSDL. He is a contributor of fundamental and technical research results to the DEVS community. His e-mail address is Hans.Vangheluwe@uantwerpen.be.