Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

AN INTRODUCTION TO MODELING AND SIMULATION WITH (PYTHON(P))DEVS

Yentl Van Tendeloo
Hans Vangheluwe

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

Romain Franceschini
University of Corsica Pasquale Paoli
UMR CNRS 6134
Campus Grimaldi
20250 Corte, France

(This is an updated version of the WSC'18 tutorial paper "Discrete Event System Specification Modeling and Simulation".)

ABSTRACT

Discrete Event System Specification (DEVS) 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 (or internal timeouts) cause instantaneous changes to the state of the system. Main advantages of DEVS are its rigorous formal definition, 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, for example in performance analysis of queueing systems. All examples are presented with the tool PythonPDEVS, though this introduction is equally applicable to other DEVS tools. We conclude with further reading on DEVS theory, DEVS variants, and DEVS tools.

1 INTRODUCTION

Discrete Event System Specification (DEVS) (Zeigler et al. 2000) 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) or internally (i.e., by the model itself due to timeouts). The next state of the system is defined based on 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 must only consider states at which events occur, skipping over all intermediate points in time. This is in contrast with discrete time models, where time is incremented with a fixed increment, and only at these times the state is updated. Discrete event models have the advantage that their time granularity can become (theoretically) unbounded, whereas time granularity is fixed in discrete time models. Nonetheless, the added complexity makes it unsuited for systems that naturally have a fixed time step.

This tutorial provides an introductory text to DEVS (often referred to as Classic DEVS) using a simple model in the domain of traffic lights. This paper is a revised version of a similar paper accompanying our tutorial at the 2017 Winter Simulation Conference Proceedings (Van Tendeloo and Vangheluwe 2017a). We start from a simple autonomous traffic light, incrementally extended up to a trafficlight with policeman interaction. Each increment serves to introduce a new component of the DEVS formalism and the corresponding (informal) semantics. It is accompagnied by an example implementation in PythonPDEVS (Van Tendeloo and Vangheluwe 2016), though the concepts are equally applicable to other tools.

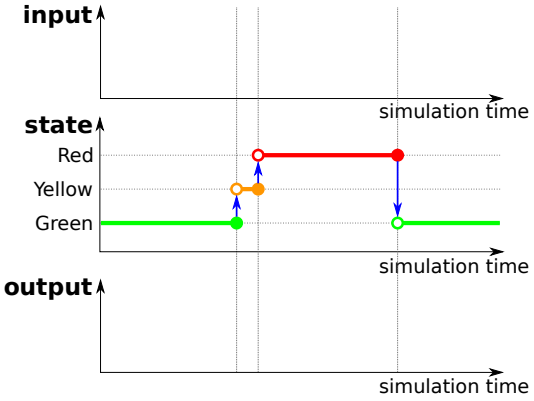

Figure 1: Trace of the autonomous traffic light.

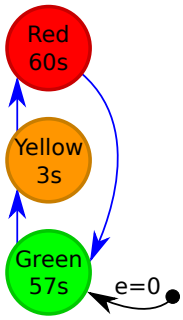

Figure 2: Model generating trace in Figure 1.

We start with atomic (i.e., non-hierarchical) models in Section 2, and introduce coupled (i.e., hierarchical) models in Section 3. Section 4 moves away from the traffic light example and presents a more complex problem. Section 5 presents several directions of further reading on DEVS. Finally, Section 6 summarizes the tutorial.

2 ATOMIC DEVS MODELS

We commence our explanation of DEVS with the atomic models. As their name suggests, atomic models are the indivisable building blocks of a model. Throughout this section, we build up the complete formal specification of an atomic model, introducing new concepts as they become required. In each intermediate step, we show and explain the concepts we introduce, how they are present in the running example, and how this influences the semantics. Next to a full specification of each increment, PythonPDEVS example code is shown to illustrate the close match.

2.1 Autonomous Model

The simplest form of a traffic light is an autonomous traffic light. Looking at it from outside, we expect to see a trace similar to that of Figure 1. Visually, Figure 2 presents an intuitive representation of a model that could generate this kind of trace.

Trying to formally describe Figure 2, we distinguish these elements:

1. **State Set** ($S : \times_{i=1}^{n} S_i$)
   The most obvious aspect of the traffic light is the state it is in, which is indicated by the three different colours it can have. These states are *sequential*: the traffic light can only be in one of these states at the same time. The set of states is not limited to enumeration style as presented here, but can contain an arbitrary number of attributes.
2. **Time Advance** ($ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$)
   For each of the states just defined, we notice the timeout in them. Clearly, some states take longer to process than others. For example, whereas we will stay in green and red a relatively long time, the time in the yellow state is only brief. This function needs to be defined for each and every element of the state set, and needs to deterministically return a duration. The duration can be any positive real number, including zero and infinity. A negative time is disallowed, as this would require simulation to go back in time. DEVS allows a time advance of exactly zero, even though this is impossible in real life. Two use cases for this exist: the delay might be very small and irrelevant to the problem we are modeling, or the state is an artificial state, without any real-world

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

Figure 3: Autonomous atomic DEVS.

```
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]
```

Figure 4: PythonPDEVS representation of Figure 3.

equivalent (e.g., as part of a design pattern). DEVS does not consider time bases, 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. Whether these units indicate seconds, years, or even π seconds, is completely up to the users, as long as it remains fixed throughout the simulation.

3. **Internal Transition ($\delta_{int} : S \rightarrow S$)**
   With the states and timeouts defined, the final part is the definition of which is the next state from a given state. This is the job of the internal transition function, which gives the next state for each state. As it is a function, every state has at most one next state, preventing any possible ambiguity. The function does not necessarily have to be total, nor injective: some states might not have a next state (i.e., if the time advance was specified as $+\infty$), and some states have the same state as next state. Up to now, only the internal transition function is described as changing the state. Therefore, it is not allowed for other functions (e.g., time advance) to modify the state: their state access is read-only.
4. **Initial Total State ($q_{init} : Q$)**
   We also need to define the initial state of the system. While this is not present in the original specification of the DEVS formalism by Zeigler et al. (2000), we include it here as it is a vital element of the model. But instead of being an "initial sequential state" (s_{init}), it is an "initial total state" (q_{init}). This means that we do not only select the initial state of the system, but also define how long we are already in this state. Elapsed time is, therefore, added to the definition of the initial total state, to allow for more flexibility when modeling a system. To the simulator, it will seem as if the model has already been in the initial state for some time. Total states are specified as follows: $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$.

We describe the model in Figure 2 as the 4-tuple defined in Figure 3. Figure 4 presents the example specification as PythonPDEVS code.

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 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.

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
```

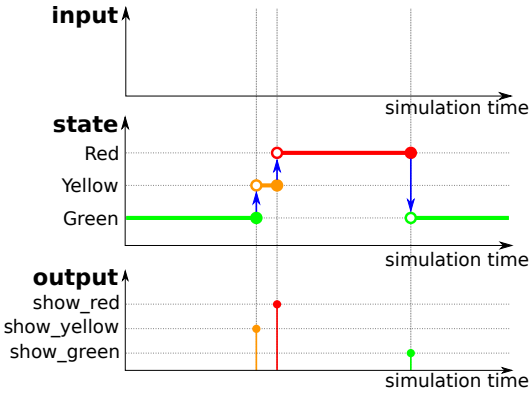

Figure 5: Trace of the autonomous traffic light with output.

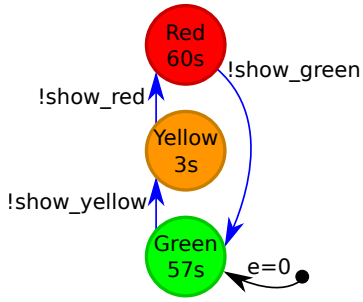

Figure 6: Model generating trace in Figure 5.

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 colour).

We, therefore, want the traffic light to output an event indicative of its current colour, 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 5. 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 6, using the exclamation mark on a transition to indicate output generation. Output only happens at an internal transition.

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

1. **Output Set** ($Y : \times_{i=1}^{l} Y_i$)
   Similarly to defining the set of allowable states, we should also define the set of allowable outputs. This set serves as an interface to other components, defining the events it expects to receive. Events can have complex attributes as well, though we again limit ourself to simple events for now. If ports are used, each port has its own output set.
2. **Output Function** ($\lambda : S \to Y \cup \{\phi\}$)
   With the set of allowable 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 no event). As seen in Figure 6, the event is generated *before* the new state is reached. This means

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

Figure 7: Autonomous atomic DEVS model with output.

```
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]}
```

Figure 8: PythonPDEVS representation of Figure 7.

that instead of the new state, the output function still uses the old state (i.e., the one that is being left). 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 colour 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't be written to, these changes need to be remembered and executed as soon as the internal transition is executed.
It is possible for the output function not to return any output, in which case it returns ϕ.

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

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.

2.3 Interruptable Model

Our current traffic light specification is still completely autonomous. While this is fine in most circumstances, the police might want to temporarily shut down the traffic lights when they are managing 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 9 shows the trace we wish to obtain. A model generating this trace is shown in Figure 10, using a question mark to indicate event reception.

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

$time \leftarrow 0$
$current\_state \leftarrow initial\_state$
$last\_time \leftarrow -initial\_elapsed$
while not termination_condition($current\_state$, $time$) **do**
  $time \leftarrow last\_time + ta(current\_state)$
  $output(\lambda(current\_state))$
  $current\_state \leftarrow \delta_{int}(current\_state)$
  $last\_time \leftarrow time$
end while

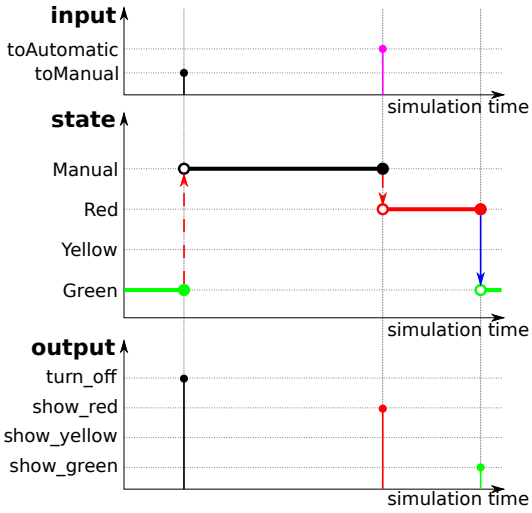

Figure 9: Trace of the autonomous traffic light.

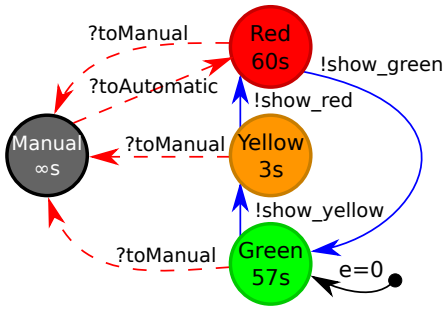

Figure 10: Naive model that should generate the trace in Figure 9 (but doesn't).

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

1. **Input Set** ($X = \times_{i=1}^{m} X_i$)
   Similar to the output set, we need to define the events we expect to receive. This is again a definition of the interface, such that others know which events are understood by this model.
2. **External Transition** ($\delta_{ext} : Q \times X \rightarrow S$)
   Similar to the internal transition function, the external transition function is allowed to define the new state as well. First and foremost, 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 (making it a total state), and the input event. The *elapsed time* indicates how long it has been for this atomic model since its 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 being the current state of the model. It is inclusive of both 0 and $ta(s)$: it is possible to receive an event right after a transition happened, or right before an internal transition happens. The combination of the current state and the elapsed time is often called the *total state* (Q) of the model. We have previously seen the total state, in the context of the initial total state.

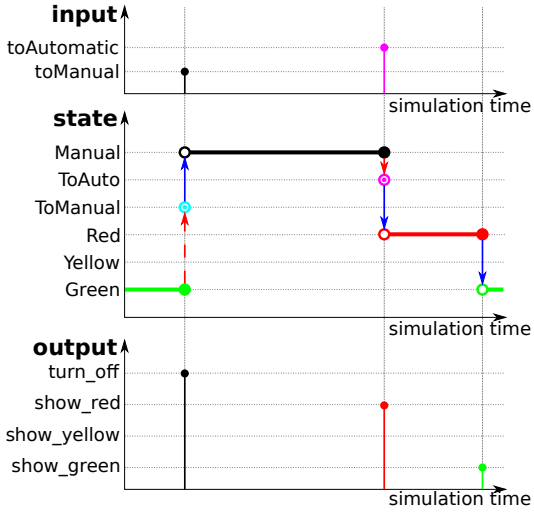

Figure 11: Trace of the interrupt traffic light with corrected artificial states.

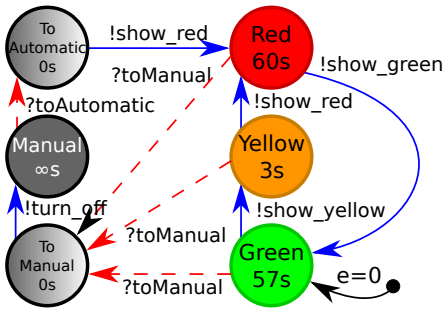

Figure 12: Model generating trace in Figure 11.

> The received event is the final parameter to this function. A new state is deterministically defined through the combination of these three parameters. Since the external transition function takes multiple parameters, multiple external transitions might be defined for a single state. Nonetheless, at most one transition should be applicable at all times to prevent non-determinism.

While we now have all elements of the DEVS specification for atomic models, we are not done yet. 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 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 11 and Figure 12 respectively.

We describe the model in Figure 12 as the 8-tuple defined in Figure 13. Figure 14 presents the example specification as PythonPDEVS code.

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. If there is an external event that must go first, we execute the external transition.

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. To combine different atomic models together and have them communicate, we introduce coupled models. This is done in the context of our previous traffic light, which will be connected to a

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

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

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

$$S = \{\textsc{Green}, \textsc{Yellow}, \textsc{Red}, \textsc{ToManual}, \textsc{ToAutomatic}, \textsc{Manual}\}$$

$$q_{init} = (\textsc{Green}, 0.0)$$

$$\delta_{int} = \{\textsc{Green} \to \textsc{Yellow}, \textsc{Yellow} \to \textsc{Red}, \textsc{Red} \to \textsc{Green}, \textsc{ToManual} \to \textsc{Manual}, \textsc{ToAutomatic} \to \textsc{Red}\}$$

$$\delta_{ext} = \{((\textsc{Green}, *), toManual) \to \textsc{ToManual}, ((\textsc{Yellow}, *), toManual) \to \textsc{ToManual}, ((\textsc{Red}, *), toManual) \to \textsc{ToManual}, ((\textsc{Manual}, *), toAutomatic) \to \textsc{ToAutomatic}\}$$

$$\lambda = \{\textsc{Green} \to show\_yellow, \textsc{Yellow} \to show\_red, \textsc{Red} \to show\_green, \textsc{ToManual} \to turn\_off, \textsc{ToAutomatic} \to show\_red\}$$

$$ta = \{\textsc{Green} \to 57, \textsc{Yellow} \to 3, \textsc{Red} \to 60, \textsc{Manual} \to +\infty, \textsc{ToManual} \to 0, \textsc{ToAutomatic} \to 0\}$$

Figure 13: 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"
```

Figure 14: PythonPDEVS representation of Figure 13.

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
```

policeman. The details of the traffic light are exactly like 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 together is the structure: how do we define a set of models and their interrelations? This is the core definition of a coupled model: it is merely a structural model that couples models together. Contrary to the atomic models, there is *no behavior* associated to a coupled model. Behaviour is the responsibility of atomic models, and structure that of coupled models.

To define the basic structure, we need three elements.

1. **Model instances (D)**
   The set of model instances defines which models are included within this coupled model.
2. **Model specifications ($\{M_i\} = \{\langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle | i \in D\}$)**
   Apart from defining the different instances of submodels, we must include the atomic model specification of these models. For each element defined in D, we include the 8-tuple specifying the atomic model. By definition, a submodel of the coupled DEVS model always needs to be an atomic model. This can be relaxed to include coupled DEVS models as well, as shown by Zeigler et al. (2000).
3. **Model influencees ($\{I_i\} = \{2^D\}$)**
   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 ($\forall i \in D : i \notin I_i$). While there is no significant problem with this in itself, it would cause the model to trigger both its internal and external transition simultaneously. As it is undefined which one should go first, this situation is not allowed. In other words, a model should not be an element in its own set of influencees.
   Second, only links within the coupled model are allowed ($\forall i \in D : I_i \subseteq D$). This is another way of saying that connections should respect modularity.

There is no explicit constraint on algebraic loops (i.e., a loop of models that have a time advance equal to zero, 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. The situation is only problematic if the circular dependency never gets resolved, causing a livelock of the simulation.

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 couples two atomic models together. But, 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** ($select : 2^D \to D$). 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 together. 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.

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

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

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

Figure 15: Coupled DEVS model of the system. Atomic DEVS models not shown for brevity.

```
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
```

Figure 16: PythonPDEVS representation of Figure 15.

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 define a coupled model as a 7-tuple. We describe the model as the 7-tuple defined in Figure 15. Figure 16 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.

4 APPLICATION TO QUEUEING SYSTEMS

The usefulness of DEVS goes further than traffic lights. 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 on their specification can be found online at https://msdl.uantwerpen.be/documentation/PythonPDEVS/queueing.html.

4.1 Problem Description

In this example, we model the behavior of a simple queue that gets served by multiple processors. Implementations of this queueing systems are widespread, such as for example at airport security. Our model is parameterizable in several ways: we can define the random distribution used for event generation times and event size, the number of processors, performance of each individual processor, and the scheduling policy of the queue when selecting a processor. Clearly, it is easier to simulate this with DEVS than it is to model mathematically. For our performance analysis, we show the influence of the number of processors (e.g., metal detectors) on the average and maximal queueing time of jobs (e.g., travellers).

Events (people) are generated by a generator using a distribution function. 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 signal the queue. The queue keeps track of available processors. When an event arrives at a processor, it is processed for some time, depending on 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.

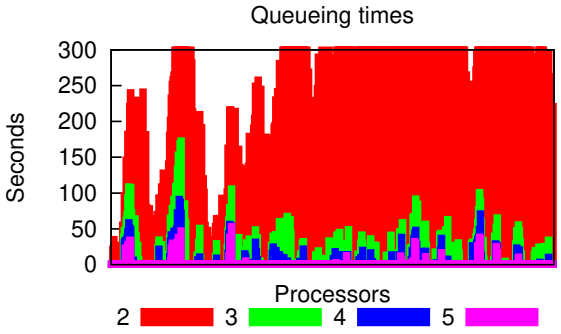

Figure 17: Evolution of queueing times for subsequent customers.

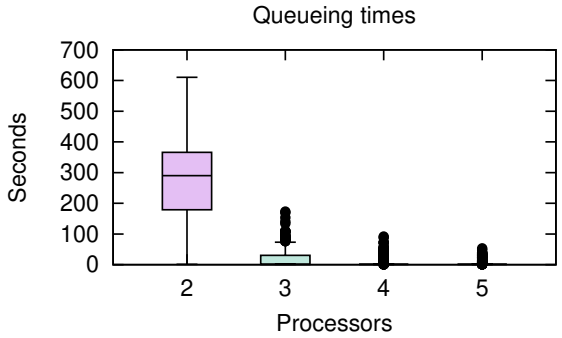

Figure 18: Boxplot of queueing times for varying number of active processors.

4.2 Performance Analysis

We now run the simulation with the models defined online at https://msdl.uantwerpen.be/documentation/PythonPDEVS/queueing.html. By executing the experiment, a CSV file is generated, which can be analyzed. Depending on the data stored during simulation, analysis can show the average queueing times, maximal queueing times, number of events, processor utilization, and so on.

Corresponding to our initial goal, we perform the simulation in order to find out the influence of opening multiple processors on the average and maximum queueing time. Figure 17 shows the evolution of the waiting time for subsequent clients. Figure 18 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 rather 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 also becomes tolerable: the average job is served immediately, with 75% of jobs being handled within 25 seconds. Further adding processors still has a positive effect on queueing times, but the effect might not warrant the increased cost in opening processors: 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. The ideal balance depends on several factors, including our model configuration and the cost function used, though these considerations are outside the scope of DEVS.

5 FURTHER READING

As this is only an introductory tutorial, we present further reading in more advanced directions. We consider three directions of interest: DEVS theory, DEVS variants, and DEVS tools.

5.1 Theory

Our introduction to DEVS has been example-driven and in a bottom-up fashion. This forced us to drop some important theoretical concepts, which have no practical use to users of DEVS.

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 rewritten as an equivalent atomic model. As the semantics of an atomic model are known, this provides denotational semantics to coupled models. The full closure under coupling proof, also called “flattening”, can be found in the original specification of DEVS (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 of DEVS (Zeigler et al. 2000).

These algorithms are only focused on specifying the behavior, and not so much 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 performance analysis and queuing simulation, DEVS is often used in other domains as well because of its genericity. 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

As the DEVS formalism is rather verbose and doesn't offer many of the features that one would expect nowadays, many variants have been created for specific problems. We provide pointers for some of them.

Parallel DEVS is probably the most popular variant, and is, in many tools, considered to be a replacement for the Classic DEVS formalism. It is mostly the same conceptually, though 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 is a dynamic structure extension to the DEVS formalism. It allows to change the structure of models during simulation, such as adding or removing new atomic or coupled models, and adding or removing links between different models. While this can also be modeled in DEVS by manually expanding the set of allowed configurations, Dynamic Structure DEVS allows for simulator support, significantly increasing performance and ease-of-use. A description of Dynamic Structure DEVS can be found in (Barros 1995), and the abstract simulator in (Barros 1998). Variants of Dynamic Structure DEVS 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 equally applicable to other DEVS simulation tools as well.

ADEVS (Nutaro 2015) is a minimalistic, though highly efficient, C++ implementation of the Parallel DEVS formalism. 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 another DEVS simulator written in Ruby.

Several comparisons are made between different tools, such as a comparison of their interface, features and performance (Van Tendeloo and Vangheluwe 2017b), and an in-depth comparison of performance (Risco-Martín et al. 2017).

6 SUMMARY

In this tutorial, we briefly presented the core ideas behind DEVS, a popular formalism for the modeling of complex dynamic systems using a discrete-event abstraction. DEVS is primarily used for the simulation of queueing networks, of which an example was given. It is most applicable for the modeling of discrete event systems with component-based modularity. It can, however, be used much more generally as a simulation assembly language, or as a theoretical foundation for these formalisms. Further reading on DEVS is provided with additional details on the theoretical aspects, a list of variations, and several tool implementations.

ACKNOWLEDGEMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO), 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 Winter Simulation Conference*, 781–785: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Barros, F. J. 1997. "Modeling Formalisms for Dynamic Structure Systems". *ACM Transactions on Modeling and Computer Simulation* 7:501–515.

Barros, F. J. 1998. "Abstract Simulators for the DSDE Formalism". In *Proceedings of the 1998 Winter simulation Conference*, 407–412: 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*, 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. San Diego, CA, USA.

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

Nutaro, James J. 2015. "adevs". http://www.ornl.gov/~1qn/adevs/. Accessed August 26th 2018.

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 – Les Journées DEVS Francophones – Théorie et Applications*, 59–66. Toulouse, France: Cépaduès.

Van Tendeloo, Y., and H. Vangheluwe. 2017a, December. "Classic DEVS Modelling and Simulation". In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, 644 – 656: IEEE.

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

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. Boca Raton, FL, USA: CRC Press.

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

AUTHOR BIOGRAPHIES

YENTL VAN TENDELOO holds a PhD (2018) from the University of Antwerp (Belgium), in the Modelling, Simulation and Design (MSDL) research lab. In his Master's thesis, he worked on MDSL's PythonPDEVS simulator, a simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS. His e-mail address is Yentl.VanTendeloo@uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium). He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results. His e-mail address is Hans.Vangheluwe@uantwerpen.be.

ROMAIN FRANCESCHINI is a post-doctoral researcher at the University of Antwerp (Belgium) in collaboration with the University of Corsica (France), where he works on modelling methods and techniques to relate multiple levels of abstraction. He received his PhD from the University of Corsica in 2017. His research interests include multi-agent systems, agent-based models and the theory of modeling and simulation. His email address is romain.franceschini@uantwerpen.be.