Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

AN INTRODUCTION TO DISCRETE-EVENT MODELING AND SIMULATION WITH DEVS

Yentl Van Tendeloo
Randy Paredis
Hans Vangheluwe

University of Antwerp – Flanders Make
Middelheimlaan 1
Antwerp, BELGIUM

ABSTRACT

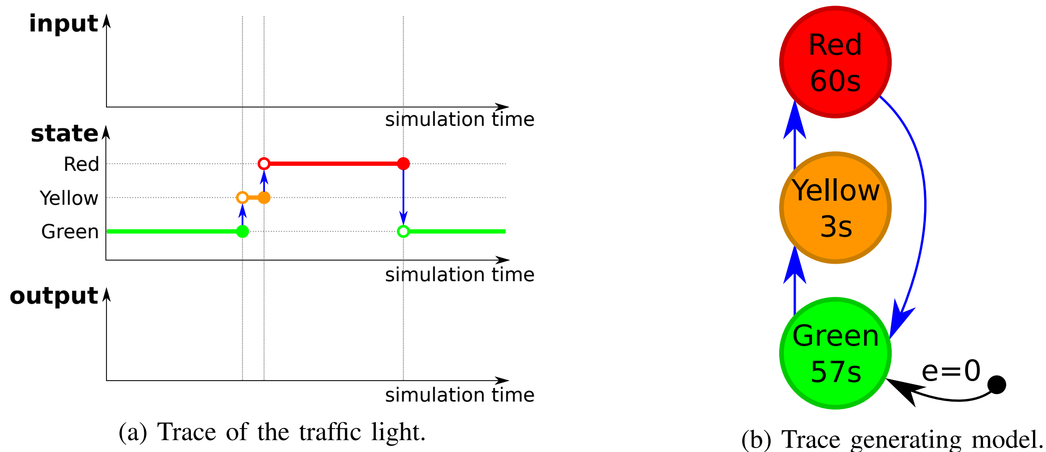
The Discrete-Event System Specification (DEVS) is a popular formalism devised by Bernard Zeigler in the late 1970s for modeling complex dynamical systems using a discrete-event abstraction. At this abstraction level, a timed sequence of pertinent "events" input to a system (or internal timeouts) causes instantaneous changes to the state of the system. The main advantages of DEVS are its precise, implementation independent specification, and its support for modular, hierarchical 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) and of Coupled (*i.e.*, hierarchical, connecting interacting components) models are introduced. Finally, a simplified DEVS model for performance analysis of vessel movements in the Port of Antwerp is presented. All examples in the paper use PythonPDEVS, though other DEVS tools could equally well be used. We conclude with suggestions for further reading on DEVS theory, variants, and tools.

1 INTRODUCTION

The Discrete-Event System Specification (DEVS) (Zeigler, Praehofer, and Kim 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 causes 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 determined 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, efficiently skipping over all intermediate points in time. This is in contrast to discrete-time models, where time is increased with a fixed increment, and the state is updated only at those times. 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 semantics would become equivalent to continuous-time semantics. The added complexity of the discrete-event abstraction makes it less appropriate (both from a cognitive, modelling point of view and from a simulation performance point of view) 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 2020 Winter Simulation Conference (Van Tendeloo, Paredis, and Vangheluwe 2020), which introduced PythonPDEVS-BBL, a Building Block Library of re-usable model components representative of commonly used concepts in industrial applications, such as queues with different queuing disciplines. A simple autonomous traffic light

979-8-3503-6966-3/23/$31.00 ©2023 IEEE



(a) Trace of the traffic light.

(b) Trace generating model.

Figure 1: Model of an autonomous traffic light.

model is incrementally extended up to a traffic light with policeman interaction. Each 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 DEVS model. 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 presents a more complex queueing problem: vessel movements in the Port of Antwerp. Section 5 presents 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. 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 one, *i.e.*, without interaction with its environment. We expect a behavior trace such as the one in Figure 1a. Figure 1b gives an intuitive visual representation of a model which generates such a trace. We distinguish the following elements in the model:

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** $ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$
   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 modeled using the time advance function ta, defined

for each and every state in 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: $e \in [0, ta(s)]$.

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 $ta(s)$. As δ_{int} is a function, every state has exactly one next state, keeping the specification deterministic. The function does not have to be injective: some states have the same state as next state. 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 a passivated 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.
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, Praehofer, and Kim (2000), it is essential to unequivocally specify a re-usable model (and re-startable simulation).

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.

For this simple formalism, the operational semantics is given in Algorithm 1.

Algorithm 1 Simulation pseudo-code for autonomous models.

$time \leftarrow 0$
$last_time \leftarrow -initial_elapsed$
$current_state \leftarrow initial_state$
while not termination_condition($current_state$, $time$) **do**
  $time \leftarrow last_time + ta(current_state)$
  $current_state \leftarrow \delta_{int}(current_state)$
  $last_time \leftarrow time$
end while

The model is initialized with simulation time set to 0, the last time that an event occurred is set to - the initial elapsed time, and the (current) state is set to the initial state (*i.e.*, GREEN). Iterative simulation is repeated as long as the simulation termination condition evaluates to false. This condition may be based on the simulated time, on the state, or on a combination of both. In the simulation loop, time is updated with

$$
\begin{aligned}
&\langle S, q_{init}, \delta_{int}, ta \rangle \\
S &= \{\textsc{Green}, \textsc{Yellow}, \textsc{Red}\} \\
q_{init} &= (\textsc{Green}, 0.0) \\
\delta_{int} &= \{\textsc{Green} \rightarrow \textsc{Yellow}, \\
&\quad \textsc{Yellow} \rightarrow \textsc{Red}, \\
&\quad \textsc{Red} \rightarrow \textsc{Green}\} \\
ta &= \{\textsc{Green} \rightarrow 57, \\
&\quad \textsc{Yellow} \rightarrow 3, \\
&\quad \textsc{Red} \rightarrow 60\}
\end{aligned}
$$

(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.

the return value of the time advance function, and the internal transition function applied to the current state gives the new state.

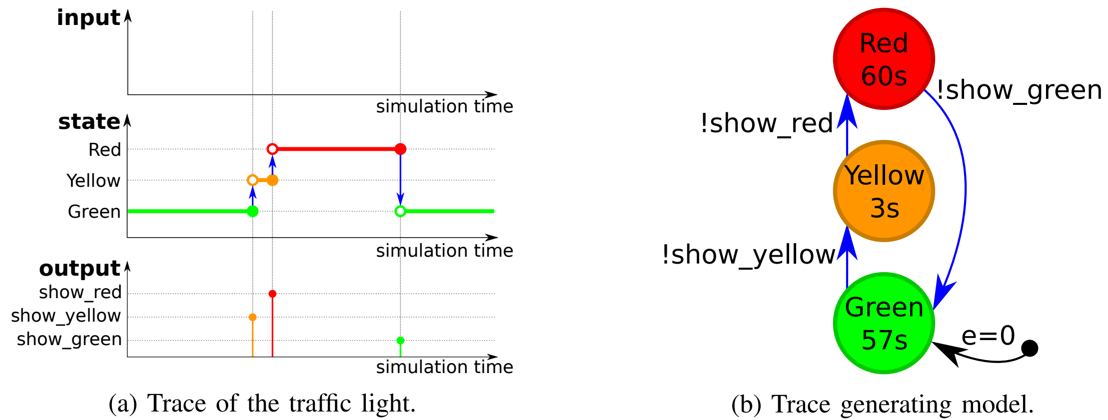
The 4-tuple in Figure 2a represents the model in Figure 1b. Figure 2b shows the corresponding PythonPDEVS code.

2.2 Autonomous Model With Output

DEVS is a modular formalism, with only the atomic model having access to its internal state. In 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. For now, this output is tightly linked to the set of states, 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. As DEVS is a discrete-event formalism, the output is only a single event at a single 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 point in time. Visually, the model is updated to Figure 3b, using the exclamation mark on a transition to denote output generation. Output only happens at the time of an internal transition.

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 of the modeled component to other components, specifying 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 λ 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



(a) Trace of the traffic light.

(b) Trace generating model.

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

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's state access is read-only. Note that in cases where the internal transition function samples from a distribution to determine the next state, determining the next state inside the output function is not straightforward.

The 6-tuple in Figure 4a represents the model in Figure 3b. Figure 4b shows the corresponding PythonPDEVS code.

The pseudo-code is now altered to include output generation, as shown in Algorithm 2. Recall that output is generated before the internal transition is executed and thus λ is invoked right before the transition.

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

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 he wishes 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 denote 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

$$\langle Y, S, q_{init}, \delta_{int}, \lambda, ta \rangle$$
$$Y = \{show_green, show_yellow, show_red\}$$
$$S = \{\textsc{Green}, \textsc{Yellow}, \textsc{Red}\}$$
$$q_{init} = (\textsc{Green}, 0.0)$$
$$\delta_{int} = \{\textsc{Green} \rightarrow \textsc{Yellow}, \textsc{Yellow} \rightarrow \textsc{Red}, \textsc{Red} \rightarrow \textsc{Green}\}$$
$$\lambda = \{\textsc{Green} \rightarrow show_yellow, \textsc{Yellow} \rightarrow show_red, \textsc{Red} \rightarrow show_green\}$$
$$ta = \{\textsc{Green} \rightarrow 57, \textsc{Yellow} \rightarrow 3, \textsc{Red} \rightarrow 60\}$$

(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.

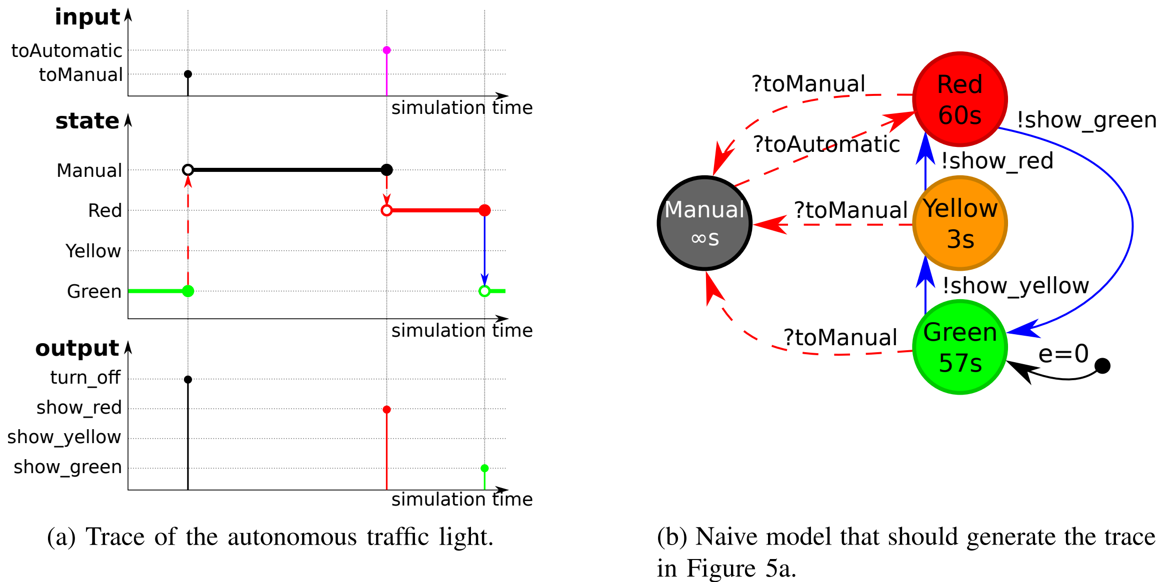
model. Events can have a complex internal structure, which is however mathematically equivalent to a flat event set. If m input ports are used, each port has its own input set X_i and $X = \times_{i=1}^{m} X_i$.

2. **External Transition Function** $\delta_{ext} : Q \times X \rightarrow S$
   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 operational semantics of an atomic model in pseudo-code. As before, we still have a simulation loop, but now external interrupts are handled. At each time step, we need to determine whether an external interrupt occurs before the time of the next internal transition. 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.

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



(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.

before an internal transition, not before an external transition. To nonetheless have an output, we need to ensure that an internal transition happens before we reach the MANUAL state. This can be done through the introduction of an artificial intermediate or *transient* 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 transient state TOMANUAL. The time advance of this state is set to 0. It is an artificial state without any meaning in the domain under study. Its output function is triggered immediately after, due to the 0 time advance, and the *turn_off* output is generated while transferring to MANUAL. Similarly, upon receiving the *toAutomatic* event, a transition to a transient TOAUTOMATIC state is needed to generate the *show_red* event. Figure 6a and Figure 6b show the corrected trace and the corresponding model. The 8-tuple in Figure 7a represents the model in Figure 6b. Figure 7b shows the corresponding 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 operating in parallel 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 Input and Output

A coupled model is a model in its own right. Just like an atomic model, it may present an interface in the form of an **input event set** X_{self} and an **output event set** Y_{self}. *self* denotes the coupled model itself. At this highest level of abstraction, a coupled model can thus be defined as a tuple: $\langle X_{self}, Y_{self} \rangle$.

3.2 Basic Coupling

Coupling the traffic light and policeman raises the question: how do we specify a set of submodels and their interactions? Contrary to the atomic models, there is *no behavior* associated with a coupled model. Behavior is the responsibility of atomic submodels, and structure that of coupled models.

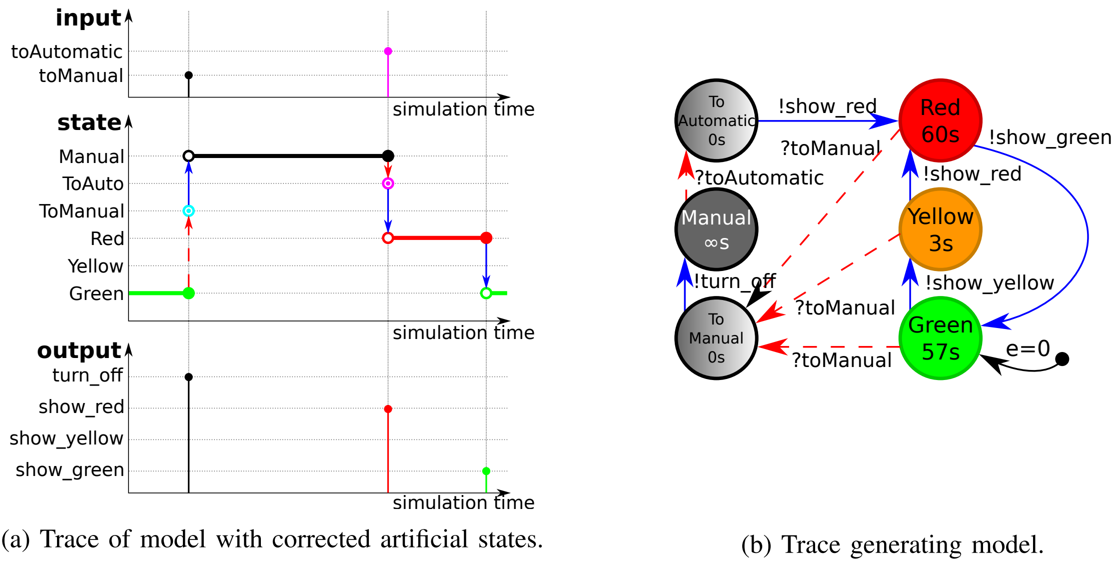
Algorithm 3 DEVS simulation pseudo-code for interruptable models.

$time \leftarrow 0$
$current_state \leftarrow initial_state$
$last_time \leftarrow -initial_elapsed$
while not termination_condition($current_state$, $time$) **do**
  $next_time \leftarrow last_time + ta(current_state)$
  if $time_next_event \leq next_time$ **then**
    $elapsed \leftarrow time_next_event - last_time$
    $current_state \leftarrow \delta_{ext}((current_state, elapsed), next_event)$
    $time \leftarrow time_next_event$
  else
    $time \leftarrow next_time$
    $output(\lambda(current_state))$
    $current_state \leftarrow \delta_{int}(current_state)$
  end if
  $last_time \leftarrow time$
end while

To define the basic structure, we need three elements.

1. **Submodel Index Set** D
   The set D of model instance descriptors referencing the submodels of the coupled model.
2. **Submodel Specifications** $\{M_i | i \in D\}$
   For each submodel index in D, we include the 8-tuple $M_i = \langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle$ specifying the atomic submodel being referred to. Here, all submodels of a coupled DEVS model are atomic models. As mentioned later in Section 5.1, any coupled model can be "flattened" to a behaviorally equivalent atomic model. Submodels may thus also be coupled models.
3. **Submodel Connection Structure: Influencee Sets** $\{I_i | i \in D \cup \{self\}\}$
   Apart from specifying the submodels, we need to specify the connections between them. To that effect, for each atomic model reference $i \in D$, we define the influencee set I_i of references to submodels influenced by that submodel (*i.e.*, submodels whose input is connected to the output of i). To allow not only for connections between submodels in D, but also between input of the coupled model $self$ and its submodels and between its submodels and output of the coupled model $self$, we add $self$ to the index set: influencee sets I_i with $i \in D \cup \{self\}$ and $I_i \subseteq D \cup \{self\}$ (or equivalently, $I_i \in 2^{D \cup \{self\}}$).
   There are restrictions on couplings. First, a submodel should not influence itself: $\forall i \in D : i \notin I_i$. This would cause the thus connected submodel to trigger both its internal and external transition simultaneously which leads to non-determinism in the precise semantics given by coupled model flattening. Second, no direct connections between inputs and outputs of $self$ are allowed: $self \notin I_{self}$. Allowing this could, as when using a time advance of 0, lead to an "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 a simulation to get stuck at a point in simulated time. In that case, the model is called *illegitimate*. Both restictions are unified in $\forall i \in D \cup \{self\} : i \notin I_i$ Note that there is no explicit constraint on algebraic loops in DEVS. Simulator implementations often to detect this (and illegitimate models in general), at simulation time, as static analysis is in general not possible.

A coupled model can thus be defined as a 5-tuple: $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\} \rangle$.



(a) Trace of model with corrected artificial states.

(b) Trace generating model.

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

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 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 very differently 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

In our example, we have full control over the design of both atomic submodels. This might not always be the case, as (atomic) models defined elsewhere might be reused. 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 given, with the policeman using *go_to_work* and *take_break* and the traffic light listening for *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.

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

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

$$\begin{aligned}
X &= \{toAutomatic, toManual\}\\
Y &= \{show_green, show_yellow, show_red,\\
&\quad turn_off\}\\
S &= \{\text{GREEN}, \text{YELLOW}, \text{RED},\\
&\quad \text{TOMANUAL}, \text{TOAUTOMATIC}, \text{MANUAL}\}\\
q_{init} &= (\text{GREEN}, 0.0)\\
\delta_{int} &= \{\text{GREEN} \rightarrow \text{YELLOW},\\
&\quad \text{YELLOW} \rightarrow \text{RED},\\
&\quad \text{RED} \rightarrow \text{GREEN},\\
&\quad \text{TOMANUAL} \rightarrow \text{MANUAL},\\
&\quad \text{TOAUTOMATIC} \rightarrow \text{RED}\}\\
\delta_{ext} &= \{((\text{GREEN}, *), toManual) \rightarrow \text{TOMANUAL}\\
&\quad ((\text{YELLOW}, *), toManual) \rightarrow \text{TOMANUAL}\\
&\quad ((\text{RED}, *), toManual) \rightarrow \text{TOMANUAL}\\
&\quad ((\text{MANUAL}, *), toAutomatic) \rightarrow \text{TOAUTOMATIC}\}\\
\lambda &= \{\text{GREEN} \rightarrow show_yellow,\\
&\quad \text{YELLOW} \rightarrow show_red,\\
&\quad \text{RED} \rightarrow show_green,\\
&\quad \text{TOMANUAL} \rightarrow turn_off,\\
&\quad \text{TOAUTOMATIC} \rightarrow show_red\}\\
ta &= \{\text{GREEN} \rightarrow 57,\\
&\quad \text{YELLOW} \rightarrow 3,\\
&\quad \text{RED} \rightarrow 60,\\
&\quad \text{MANUAL} \rightarrow +\infty,\\
&\quad \text{TOMANUAL} \rightarrow 0,\\
&\quad \text{TOAUTOMATIC} \rightarrow 0\}
\end{aligned}$$

(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.

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

With the addition of this final element, 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.

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

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

(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 of the system. Atomic DEVS models are not shown for brevity.

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

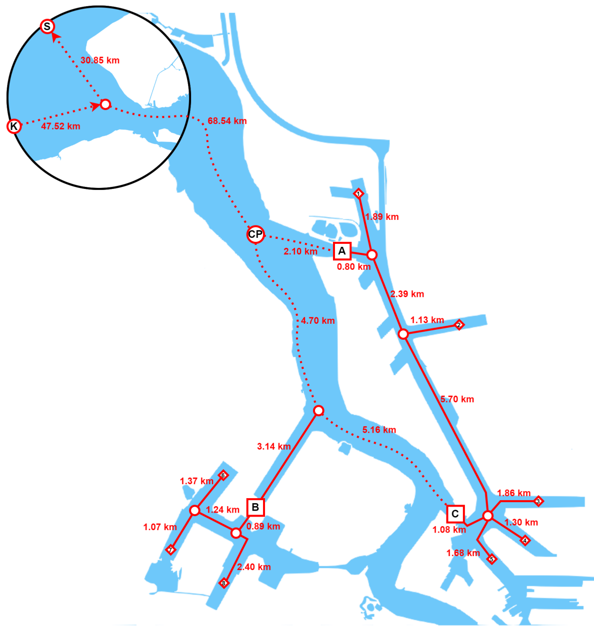
4 APPLICATION TO THE PORT OF ANTWERP

DEVS is highly suited to model systems whose behavior involves competition for shared resources. This leads to queues of entities waiting for a shared resource to become available. For the purpose of this example, we present a simplified model of the Port of Antwerp (PoA) in which vessels travel from the North Sea, over the Scheldt river, through canals and locks toward docks, and back. These canals, locks and docks have a finite capacity and we choose to model them as (active, i.e., they exhibit behavior) resources. We choose to model arrival and departure of vessels at different points in the harbor as (passive, i.e., they do not exhibit behavior and are only generated, sent and received) events. The full specification of the PoA example is found following the permanent link http://msdl.uantwerpen.be/people/hv/teaching/MoSIS/202223/assignments/DEVS. Here, we only summarize the essence.

4.1 Problem Description

Our model is parameterizable in several ways: we can define the distributions used for event generation: vessel inter-arrival times and vessel properties (*i.e.*, velocity, size, type, ...), performance (service times) of each individual lock, and the scheduling policy for access to the locks. Given the complexity of the problem, both in structure and distributions, mathematical modeling and subsequent analytical solution is rarely feasible. That is why we build a DEVS simulation model. The Property of Interest for performance analysis, is the influence of the lock behavior on the average and maximal queueing time of vessels.

Figure 9a shows an abstraction of the full port as a coupled DEVS model depicted by the nodes and edges of the graph overlaying the waterways. Figure 9c shows a code snippet for this coupled model in PythonPDEVS. Events (vessel arrivals) are produced by a generator (marked “K” in the figure) based on an Inter-Arrival Time distribution (which alters slightly every hour). They travel over `River` sections (depicted by dotted lines/arrows, where overtaking other vessels is allowed) until they reach a desired `Lock` (depicted by squares). The `Lock` works First-In-First-Out (FIFO), as long as there is room for another vessel. If the first vessel cannot enter the `Lock` as the maximum capacity of the lock is reached, the next vessel (which may be smaller and still fits in the lock) is tried. Behind the `Locks` are `Canals` (depicted



(a) Visualization of the Port of Antwerp example.

```
class Sea(AtomicDEVS):
  def __init__(self, name):
    super(Sea, self).__init__(name)
    self.ship_in = self.addInPort("ship_in")
    self.state = []

  def extTransition(self, inputs):
    if self.ship_in in inputs:
      self.state.append(inputs[self.ship_in])
    return self.state
```

(b) Sea atomic DEVS in PythonPDEVS.

```
class Port(CoupledDEVS):
  def __init__(self):
    super(Network, self).__init__("Port")

    self.generator = self.addSubModel(
      Generator())
    self.anchorpoint = self.addSubModel(
      Anchorpoint())
    self.dock_1 = self.addSubModel(Dock(1))

    ...

    self.connectPorts(self.generator.ship_out,
        self.anchorpoint.ship_in)

    ...
```

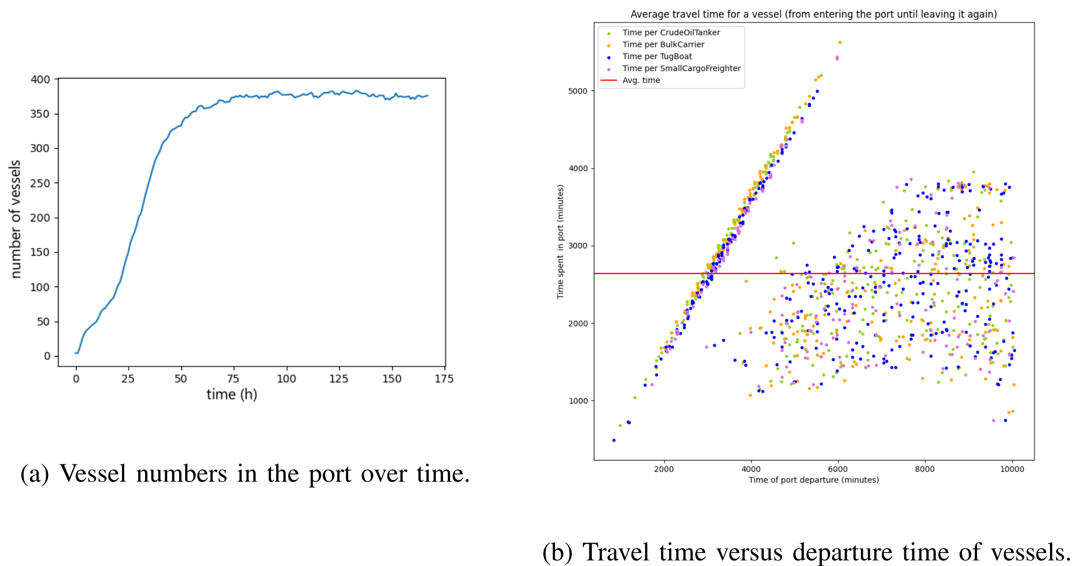
(c) part the full port coupled DEVS in PythonPDEVS.

Figure 9: Modeling the Port of Antwerp (PoA).

by full lines; same as a `River`, but overtaking is not allowed) that will eventually lead to `Docks` (depicted by diamonds). At a `Dock`, the vessels can wait for a desired time (determined by the loading/unloading time) before departing again in the opposite direction. To help the vessels "navigate" the port, so-called `Confluences` (depicted by circles) are added on merging/diverging rivers and canals. Thus, we can model how each vessel follows its own predefined path through the port. The `Sea` (marked "S" in the figure) is a collector for all vessels, allowing statistical information gathering. Its PythonPDEVS code is given in figure 9b.

4.2 Performance Analysis

We now perform a simulation experiment for a given set of parameters (given in the online detailed and full description) and study the results. Our goal was to find out the influence of the number of vessels on the travel time. Figure 10a shows how many vessels are in the port at each hour of the simulation. The port starts empty and gradually fills up until it reaches steady state. This is a typical simulation by-product called the "warm-up period". Statistics should only be recorded after this transient period is over. Figure 10b shows the travel time of vessels as a function of the departure time from the port. The slanted "line" (very small variance) on the left hand side is due to the warm-up period. When the port operation is in steady state, the distribution of travel times is quite smeared out, and seems to broaden with time. This effect deserves further investigation.



(a) Vessel numbers in the port over time.

(b) Travel time versus departure time of vessels.

Figure 10: Analyzing simulation results.

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 most 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 a *denotational semantics* for coupled models. The description of closure under coupling, also called “flattening”, can be found in (Zeigler, Praehofer, and Kim 2000) and (Van Tendeloo and Vangheluwe 2020). 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. 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 (Chow and Zeigler 1994) 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*. Dynamic Structure DEVS (DSDEVS) (Barros 1995; Barros 1998)

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. Variants of DSDEVS have also been created for Parallel DEVS (Barros 1997), and with different ways of representing the dynamicity (Uhrmacher 2001). Cell-DEVS (Wainer and Giambiasi 2001) 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. All concepts introduced in this tutorial are however applicable to other DEVS simulation tools as well. ADEVS (Nutaro 2010) is a minimalistic, highly efficient, C++ implementation of the Parallel DEVS formalism. A recent C++-based addition is Cadmium (Belloli, Vicino, Martin, and Wainer 2019), a C++17 header-only Parallel DEVS simulator. DEVS-Suite (Kim, Sarjoughian, and Elamvazhuthi 2009) is a full modeling and simulation environment implemented in Java, with a visual simulation interface. DEVSimPy (Capocchi, Santucci, Poggi, and Nicolai 2011) is a modeling and simulation tool with PythonPDEVS as its simulation kernel. A Parallel DEVS debugging extension (Van Mierlo, Van Tendeloo, and Vangheluwe 2017) allows for fine-grained debugging, based on the PythonPDEVS simulation kernel. DesignDEVS (Goldstein, Breslav, and Khan 2016) is an intuitive DEVS simulator. DEVS-Ruby (Franceschini, Bisgambiglia, Bisgambiglia, and Hill 2014) is a DEVS simulator written in Ruby. Several comparisons exist between different tools, based on their interface, features and performance (Van Tendeloo and Vangheluwe 2017), and an in-depth comparison of performance (Risco-Martín, Mittal, Fabero Jiménez, Zapater, and Hermida Correa 2017).

6 SUMMARY

This tutorial briefly presented the core concepts of the discrete-event formalism DEVS in a bottom-up, incremental fashion. An important use of DEVS is simulation-based performance analysis of queueing networks. A simplified model of the Port of Antwerp was used as an example. Further references on DEVS were provided for more information on the theoretical aspects, for a list of variants, and supporting tools.

ACKNOWLEDGEMENTS

We acknowledge partial support for this work by Flanders Make, the strategic research center for the manufacturing industry in Flanders.

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.

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., R. Paredis, and H. Vangheluwe. 2020. "An Introduction To Modular Modeling And Simulation With PythonPDEVS And The Building-Block Library PythonPDEVS-BBL". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 1152–1166. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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., and H. Vangheluwe. 2020. "DEVS: Discrete-Event Modelling and Simulation for Performance Analysis of Resource-Constrained Systems". In *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*, edited by P. Carreira, V. Amaral, and H. Vangheluwe, Chapter 5, 127–153. Springer.

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., and N. Giambiasi. 2001. "Discrete event modeling and simulation technologies". Chapter Timed cell-DEVS: modeling and simulation of cell spaces, 187–214. Springer.

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 PythonPDEVS in his 2014 Masters thesis. His e-mail address is Yentl.VanTendeloo@uantwerpen.be.

RANDY PAREDIS is a Ph.D. student in the MSDL. He explores a product family framework and generic architecture for model-based design of Digital Twins. He also develops and combines modeling and simulation languages by mapping them onto the DEVS formalism. His e-mail address is randy.paredis@uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium) where he heads the MSDL and an Adjunct Professor at McGill University (Canada). He is a long-time contributor of fundamental and technical research results to the DEVS community. His overall research interest is in Multi-Paradigm Modelling (MPM) of complex, software-intensive, Cyber-Physical Systems (CPS). His e-mail address is Hans.Vangheluwe@uantwerpen.be.