INTRODUCTION TO PARALLEL DEVS MODELLING AND SIMULATION

Yentl Van Tendeloo
University of Antwerp, Belgium
Yentl.VanTendeloo@uantwerpen.be

Hans Vangheluwe
University of Antwerp, Belgium
Flanders Make, Belgium
McGill University, Montréal, Canada
hv@cs.mcgill.ca

ABSTRACT

DEVS is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. Main advantages of DEVS are its rigorous formal definition, and its support for modularity: models can be hierarchically nested. Thanks to these properties, DEVS frequently serves as a simulation "assembly language" to which models in other formalisms are mapped. This makes it possible to combine models in different formalisms together by mapping both to DEVS. This tutorial introduces the practical use of the Parallel DEVS formalism in a bottom-up fashion. We start from simple autonomous Atomic (i.e., non-hierarchical) DEVS models and increment up to Coupled (i.e., hierarchical) DEVS models. Each increment is illustrated with a minimal running example. The focus is on the practical use of DEVS modelling and simulation, though necessary theoretical foundations are interleaved. Examples are presented using PythonPDEVS, though the foundations and techniques apply to other DEVS simulation tools as well.

Keywords: DEVS, Parallel DEVS, discrete event simulation

1 INTRODUCTION

This tutorial provides an introductory tutorial on the practical usage of the Parallel DEVS formalism (Chow and Zeigler 1994). Parallel DEVS is a discrete-event formalism with a long history, which started in 1976 with the introduction of Classic DEVS (Zeigler 1976). The content of this tutorial is primarily based on the influential book "Theory of Modelling and Simulation" (Zeigler 1976), which defined the original Classic DEVS formalism, with selected topics from "Multifacetted Modelling and Discrete Event Simulation" (Zeigler 1984) and the more recent edition of "Theory of Modelling and Simulation" (Zeigler, Praehofer, and Kim 2000). For the extension to Parallel DEVS, the founding papers of Parallel DEVS (Chow and Zeigler 1994, Chow, Zeigler, and Kim 1994) are integrated.

Contrary to other introductions to Parallel DEVS, our tutorial provides a bottom-up introduction that focuses on example-driven explanation of the various aspects of Parallel DEVS. And while we still briefly touch upon the theoretical foundations, these are interleaved with an actual application of the theory. This application is in the form of a traffic light: we start from a simple autonomous traffic light without output and keep extending this to use all features of Parallel DEVS. Afterwards, hierarchy is introduced with the same example, and we glimpse at more advanced concepts.

As our tutorial is example-driven, we use PythonPDEVS (Van Tendeloo and Vangheluwe 2014, Van Tendeloo and Vangheluwe 2016) as a simple Parallel DEVS simulator written in Python, a high-level, interpretted language. We use PythonPDEVS instead of another Parallel DEVS simulator, as Python is a simple language that is familiar to most people, and its semantics is, at least in the scope of this tutorial, intuitive. This

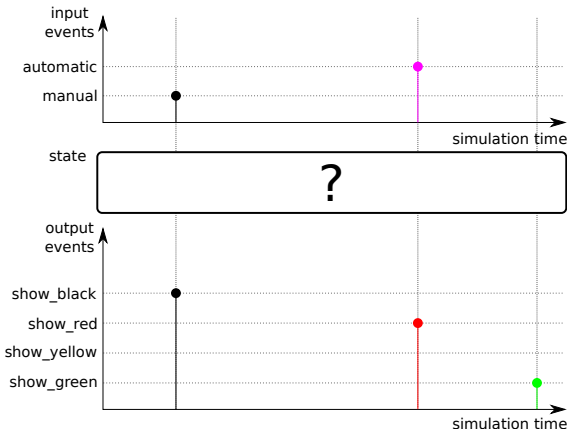


Figure 1: Discrete event abstraction.

makes PythonPDEVS models easy to understand, even without intimate knowledge of Python. Nonetheless, our tutorial focuses primarily on the theoretical aspects of Parallel DEVS, which is equally valid for other Parallel DEVS simulation tools. A comparison of several popular DEVS tools (Van Tendeloo and Vangheluwe 2017) can be used to find the most appropriate tool. PythonPDEVS and all used examples can be found online at the PythonPDEVS project page (http://msdl.cs.mcgill.ca/projects/PythonPDEVS/), which also links to the documentation.

2 DISCRETE EVENT MODELLING

DEVS is a discrete-event formalism, so we start of with a brief description of the discrete-event abstraction. In a discrete-event abstraction, only a finite number of relevant events occur during a bounded time interval. This is in contrast to continuous models, which take *continuous signals* as input.

A discrete-event system is characterized by the output events it generates in response to a series of input events. There is no restriction on the internal state of the model, as the system is seen as a "black box".

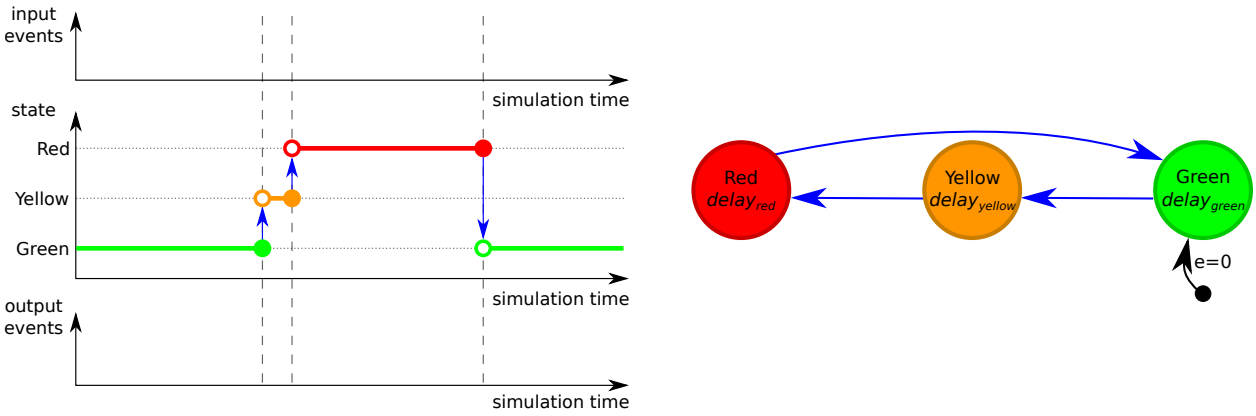
An example of such a trace is shown in Figure 1, representing a traffic light system. As can be seen, there is a finite number of input and output events in any bounded time interval. It is not surprising that the input is a series of events, as it is someone pressing a button to switch the mode of the traffic light. The output, however, also uses events, and thus only raise an event when the color of the traffic light changes. The receiver of the event therefore needs to remember the last event it received, in order to tell what light is currently active. As there is no restriction on the internal state, this part is hidden in the figure.

In the remainder of the tutorial, we go in-depth as to how we can define a discrete event system that generates this output trace for this specific input trace.

3 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. It is these blocks that define behaviour. Nonetheless, they have no hierarchy whatsoever.

Throughout this section, we build up the complete formal specification of an atomic model, introducing new concepts when they become necessary. In each intermediate step, we show and explain the concepts we introduce, how they are present in the running example, and how this influences semantics.



(a) Trace of the autonomous traffic light.

(b) Model generating trace in Figure 2a.

Figure 2: Simple autonomous traffic light.

3.1 Autonomous Model

The simplest form of a traffic light is an autonomous traffic light. We expect to see a trace similar to that of Figure 2a. Visually, Figure 2b presents an intuitive representation of a model that could generate this kind of trace.

The atomic model can take external parameters, passed at initialization time. For example, it is possible for the atomic DEVS model to use parameters $delay_{green}$, $delay_{yellow}$, and $delay_{red}$ instead of defining constants. These parameters can then be used in the model as if they were constants.

Trying to formally describe Figure 2b, 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 is in exactly one of these states at all times.
2. **Time Advance** ($ta : S \to \mathbb{R}^{+}_{0,+\infty}$)
   For each of the states just defined, we notice the timeout in them: the time advance of the state. This duration can be any positive real number, including zero and infinity. Note that DEVS does not consider time bases: simulation time is just a real number and time units are left up to the user.
3. **Internal Transition** ($\delta_{int} : S \to S$)
   With the states and timeouts defined, the final part is the definition of which is the next state from a given state: the internal transition. For each state with a finite time advance, it returns the next state upon expiration of the time advance timeout.
4. **Initial Total State** ($q_{init} : Q = (s,e) | s \in S, 0 \leq e \leq ta(s)$)
   We also define the initial state of the system (Van Tendeloo and Vangheluwe 2018), even though it is not present in the original specification of the Parallel DEVS formalism. Nonetheless, it is a vital part of any model. But instead of being an "initial state (s_{init})", it is a total state. This means that we not only select the initial state of the system, but also how long we are already in this state: the *elapsed time*.

We describe the model in Figure 2b as a 4-tuple in Figure 3, and as PythonPDEVS code in Figure 4.

$$M = \langle S, q_{init}, \delta_{int}, ta \rangle$$

$$\begin{aligned} S &= \{\text{GREEN}, \text{YELLOW}, \text{RED}\} \\ q_{init} &= (\text{GREEN}, 0.0) \\ \delta_{int} &= \{\text{GREEN} \to \text{YELLOW}, \\ &\quad \text{YELLOW} \to \text{RED}, \\ &\quad \text{RED} \to \text{GREEN}\} \\ ta &= \{\text{GREEN} \to delay_{green}, \\ &\quad \text{YELLOW} \to delay_{yellow}, \\ &\quad \text{RED} \to delay_{red}\} \end{aligned}$$

Figure 3: Specification of an autonomous traffic light.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
            delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

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

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red}[self.state]
```

Figure 4: Implementation of an autonomous traffic light.

For this simple formalism, we define the semantics as in Algorithm 1. The model is initialized with the simulation time set to 0, the state set to the initial state (*e.g.*, GREEN), and the time of the last transition set to the initial elapsed time (*e.g.*, 0.0). 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.

Algorithm 1 DEVS simulation pseudo-code for autonomous models.

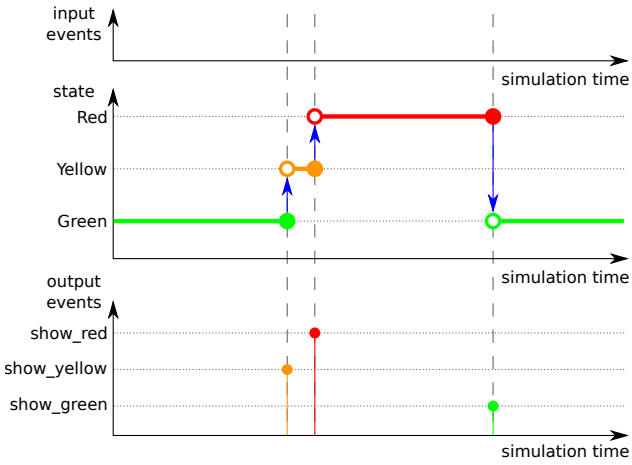
```
time ← 0
current_state ← initial_state
last_time ← -initial_elapsed
while not termination_condition() do
   time ← last_time + ta(current_state)
   current_state ← δ_int(current_state)
   last_time ← time
end while
```

3.2 Autonomous Model with Output

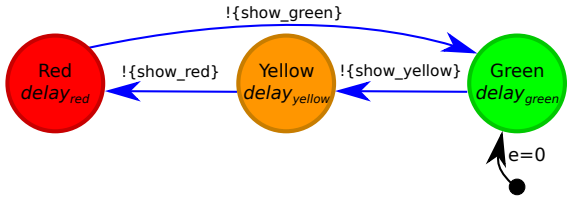
Recall that DEVS is a modular formalism, with only the atomic model having access to its internal state. This raises a problem for our traffic light: others have no way of knowing its current state (*i.e.*, its colour).

We therefore want the traffic light to output its colour, such that others can use it. Our desired trace is shown in Figure 5a. We see that we now output events indicating the start of the specified period. DEVS is a discrete event formalism: the output is only a single event indicating the 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. This model is shown in Figure 5b, using the exclamation mark on a transition to indicate output generation. Our event is enclosed in curly braces, as we output a bag of events instead of a single event.

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



(a) Trace of the autonomous traffic light with output.



(b) Model generating trace in Figure 5a.

Figure 5: Simple autonomous traffic light with 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 the model expects to receive.
2. **Output Function** ($\lambda : S \to Y^b$)
   With the set of allowable events defined, we still need a function to actually generate the events. As seen in the Figure of the model, the event is generated *before* the new state is reached. This means that instead of the new state, the output function still uses the old state (*i.e.*, the one that is being left). In the case of our traffic light, the output function needs to return the light to toggle, based on 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, as it is entering the YELLOW state. We output a bag of events, meaning that, at the same time, multiple events can be raised simultaneously, potentially even the same event. This bag, however, can also be empty, indicating that the output function did not generate any output for this state.

We describe the model in Figure 5b as a 6-tuple in Figure 6, and as PythonPDEVS code in Figure 7.

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

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

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

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

$$\begin{aligned}
Y &= \{show_green, show_yellow, show_red\} \\
S &= \{\text{GREEN}, \text{YELLOW}, \text{RED}\} \\
q_{init} &= (\text{GREEN}, 0.0) \\
\delta_{int} &= \{\text{GREEN} \to \text{YELLOW}, \\
&\quad \text{YELLOW} \to \text{RED}, \\
&\quad \text{RED} \to \text{GREEN}\} \\
\lambda &= \{\text{GREEN} \to \{show_yellow\}, \\
&\quad \text{YELLOW} \to \{show_red\}, \\
&\quad \text{RED} \to \{show_green\}\} \\
ta &= \{\text{GREEN} \to delay_{green}, \\
&\quad \text{YELLOW} \to delay_{yellow}, \\
&\quad \text{RED} \to delay_{red}\}
\end{aligned}$$

Figure 6: Specification of an autonomous traffic light with output.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
            delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

        self.output = self.addOutPort("Output")

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

    def outputFnc(self):
        if self.state == "Green":
            return {self.output: ["show_yellow"]}
        elif self.state == "Yellow":
            return {self.output: ["show_red"]}
        elif self.state == "Red":
            return {self.output: ["show_green"]}
        return {}

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red}[self.state]
```

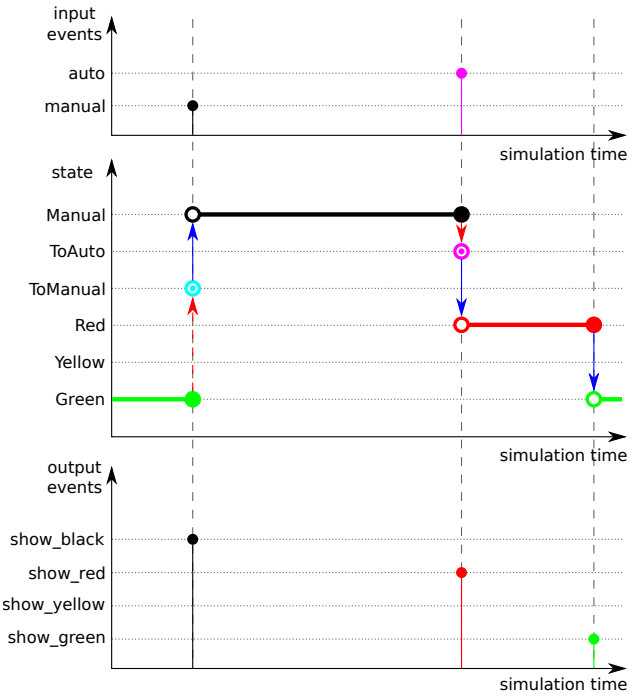
Figure 7: Implementation of an autonomous traffic light with output.

3.3 Interruptable Model

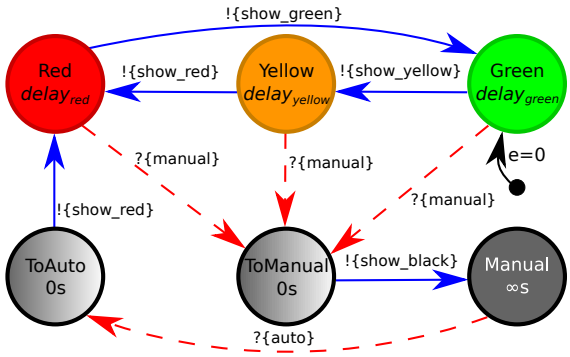
Our current traffic light specification is still completely autonomous. The police might want to temporarily shut down the traffic lights when they are manually directing traffic. To allow for this, our traffic light must process external incoming events: the event from the policeman to shutdown and to start up again. Figure 8a shows the trace we wish to obtain. A model generating this trace is shown in Figure 8b, using a question mark to indicate event reception. Similar to event generation, events are received in the form of a bag of events. As it is a bag, the order of events is non-deterministic, multiple events can be received simultaneously, and events can arrive multiple times at the same time.

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** ($\delta_{ext} : Q \times X^b \to S$)
   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 and the bag of input events. The *elapsed time* indicates how long it has been for this atomic model since the last transition (either internal or external), similarly to how it was defined in q_{init}.



(a) Trace of the interruptable traffic light.



(b) Model generating trace in Figure 8a.

Figure 8: Interruptable traffic light.

Note the special structure around the MANUAL state. The output function was defined as being called before an internal transition only. As such, external transitions do not raise events. Therefore, additional transient states TOMANUAL and TOAUTO need to be added, with a time advance of zero, on which internal transition the output events will be raised.

Finally, a corner case still needs to be covered: it is possible for the atomic model to receive an input event at exactly the same time as an internal transition was scheduled. The question is then which transition to take: the internal or external transition?

Parallel DEVS solves this by introducing the **confluent transition function** (δ_{conf}): when both the internal and external transition function are to be executed, the confluent transition function is executed instead. Often, the δ_{conf} is defined as first invoking the internal transition, followed by the external transition. Its definition is a mix of both: $\delta_{conf} : S \times X^b \to S$.

We describe this model as a 9-tuple in Figure 9, and as PythonPDEVS code in Figure 10.

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 is scheduled before the internal interrupt. If that is not the case, we simply act as before, by executing the internal transition. If there is an external event that must go first, we execute the external transition.

4 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 and couple atomic models together, we introduce coupled models. We

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

$$\begin{aligned}
X &= \{auto, manual\} \\
Y &= \{show_green, show_yellow, \\
&\quad show_red, show_black\} \\
S &= \{\text{GREEN}, \text{YELLOW}, \text{RED}, \\
&\quad \text{TOMANUAL}, \text{TOAUTO}, \text{MANUAL}\} \\
q_{init} &= (\text{GREEN}, 0.0) \\
\delta_{int} &= \{\text{GREEN} \to \text{YELLOW}, \\
&\quad \text{YELLOW} \to \text{RED}, \\
&\quad \text{RED} \to \text{GREEN}, \\
&\quad \text{TOMANUAL} \to \text{MANUAL}, \\
&\quad \text{TOAUTO} \to \text{RED}\} \\
\delta_{ext} &= \{((_,_), \{manual\}) \to \text{TOMANUAL} \\
&\quad ((\text{MANUAL}, _), \{auto\}) \to \text{TOAUTO}\} \\
\delta_{conf} &= \{(s,x) \to \delta_{ext}((\delta_{int}(s), 0), x)\} \\
\lambda &= \{\text{GREEN} \to \{show_yellow\}, \\
&\quad \text{YELLOW} \to \{show_red\}, \\
&\quad \text{RED} \to \{show_green\}, \\
&\quad \text{TOMANUAL} \to \{show_black\}, \\
&\quad \text{TOAUTO} \to \{show_red\}\} \\
ta &= \{\text{GREEN} \to delay_{green}, \\
&\quad \text{YELLOW} \to delay_{yellow}, \\
&\quad \text{RED} \to delay_{red}, \\
&\quad \text{MANUAL} \to +\infty, \\
&\quad \text{TOMANUAL} \to 0, \\
&\quad \text{TOAUTO} \to 0\}
\end{aligned}$$

Figure 9: Specification of an interruptable traffic light.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
            delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

        self.interrupt = self.addInPort("Interrupt")
        self.output = self.addOutPort("Output")

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

    def extTransition(self, event):
        if "manual" in event[self.interrupt]:
            return "ToManual"
        elif "auto" in event[self.interrupt]:
            if self.state == "Manual":
                return "ToAuto"
        return self.state

    def confTransition(self, event):
        self.state = self.intTransition(self)
        self.elapsed = 0.0
        return self.extTransition(self, event)

    def outputFnc(self):
        if self.state == "Green":
            return {self.output: ["show_yellow"]}
        elif self.state == "Yellow":
            return {self.output: ["show_red"]}
        elif self.state == "Red":
            return {self.output: ["show_green"]}
        elif self.state == "ToManual":
            return {self.output: ["show_black"]}
        elif self.state == "ToAuto":
            return {self.output: ["show_red"]}
        return {}

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red,
                "Manual": INFINITY,
                "ToManual": 0,
                "ToAuto": 0}[self.state]
```

Figure 10: Implementation of an interruptable traffic light.

Algorithm 3 DEVS simulation pseudo-code for interruptable models.

```
time ← 0
current_state ← initial_state
last_time ← -initial_elapsed
while not termination_condition() 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 if time_next_event > next_time then
    time ← next_time
    output(λ(current_state))
    current_state ← δ_int(current_state)
  else if time_next_event = next_time then
    time ← next_time
    output(λ(current_state))
    current_state ← δ_conf(current_state, next_event)
  end if
  last_time ← time
end while
```

do this in the context of our previous traffic light, which is connected to a policeman. The details of the traffic light are exactly like before. the details of the policeman are irrelevant here, with the exception of its interface: we know that it raises the *take_break* and *go_to_work* events.

4.1 Basic Coupling

Contrary to the atomic models, there is no behaviour whatsoever associated to a coupled model. Behaviour is the responsibility of atomic models, and structure that of coupled models. Therefore, to a coupled model, we only need connection concepts, such as submodels and connections.

We specify a simple coupled model as a 3-tuple $\langle D, \{M_i\}, \{I_i\} \rangle$.

1. **Model instances (D)**
   The set of model instances defines which models are included within this coupled model.
2. **Model specifications ($\{M_i\}$)**
   For each element defined in D, we include the 9-tuple specifying the atomic model. By definition, a submodel of the coupled DEVS model always needs to be an atomic model. Through closure under coupling, as mentioned later, we show that this can be extended to coupled models as well.

$$\{M_i\} = \{\langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \delta_{conf,i}, \lambda_i, ta_i \rangle | i \in D\}$$

3. **Model influencees ($\{I_i\}$)**
   The next step is the connection of submodels, which is defined through influencee sets. For each atomic model, its influencee set defines the models affected by its output. There are some constraints on couplings, to prevent invalid models:
   - A model should not influence itself. ($\forall i \in D : i \notin I_i$)

•Only links within the coupled model are allowed. ($\forall i \in D : I_i \subseteq D$)

4.2 Input and Output

Our coupled model now couples two atomic models together. And 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 to an entity external to this coupled model. We augment the coupled model with input and output events, which serve as the interface to the coupled model. This adds the components X_{self} and Y_{self} to the tuple, respectively the set of input and output events, resulting in a 5-tuple $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}\rangle$.

The constraints on the couplings are relaxed to accomodate for this change: a model can be influenced by the input of the coupled model, and likewise the models can also influence the output events of the coupled model. The previously defined constraints are relaxed to allow for $self$, the coupled model itself:

- A model should not influence itself. ($\forall i \in D \cup \{self\} : i \notin I_i$)
- Only links within the coupled model are allowed. ($\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$)

4.3 Translation Functions

Recall that our traffic light listened to the *manual* and *auto* events. The policeman, however, outputs *go_to_work* and *take_break* events. This mismatch prevents them from being coupled directly, as $Y_{police} \neq X_{light}$.

To handle this problem, connections are augmented with a **translation function** ($Z_{i,j}$), which specifies a translation of output to input events.

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

These translation functions are defined for each connection, including those between the coupled model's input and output events.

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$$

The translation function is implicitly assumed to be the identity function if it is not defined. In case an event needs to traverse multiple connections, all translation functions are chained in order of traversal.

With the addition of this final element, we define a coupled model as a 6-tuple.

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

4.4 PythonPDEVS

Implementing a Coupled DEVS model in PythonPDEVS is simple, as it merely instantiates the Atomic DEVS models and connects them. Of course, it is also possible for the submodels to be coupled DEVS models themselves. An example for the coupling of a policeman with a trafficlight is shown in Figure 11.

```
def z_function(evt_in):
    if evt_in == "take_break":
        return "auto"
    elif evt_in == "go_to_work":
        return "manual"

class TrafficLightSystem(CoupledDEVS):
    def __init__(self):
        CoupledDEVS.__init__(self, "TrafficLightSystem")
        self.light = self.addSubModel(TrafficLight(57, 3, 60))
        self.police = self.addSubModel(PoliceMan())

        self.connectPorts(self.police.output, self.light.interrupt, z_function)
```

Figure 11: Implementation of a Coupled DEVS model.

4.5 Closure Under Coupling

Similar to atomic models, we need to define the semantics of coupled models. But instead of explaining the semantics from scratch, by defining some pseudo-code, we map coupled models to equivalent atomic models. Semantics of a coupled model is thus defined in terms of an atomic model. In addition, this flattening removes the constraint of coupled models that their submodels should be atomic models: if a coupled model is a submodel, it can be flattened to an atomic model, thus matching the definition.

In essence, for any coupled model specified as $CM =< X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$, we define an equivalent atomic model specified as $< X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$. A detailed discussion of this is omitted, as it is not essential to model using Parallel DEVS. Closure under coupling is explained in the original Parallel DEVS specification (Chow and Zeigler 1994), and the necessary changes for the q_{init} are presented separately (Van Tendeloo and Vangheluwe 2018).

4.6 Abstract Simulator

Actually performing closure under coupling at runtime to provide semantics is an inefficient option. Therefore, an equivalent semantic mapping is defined in the form of operational semantics by the abstract simulator (Chow, Zeigler, and Kim 1994). A detailed discussion of the abstract simulator is omitted as well.

5 SUMMARY

In this tutorial, we have considered Atomic DEVS models and Coupled DEVS models through the use of a simple running example: a traffic light. Atomic DEVS models consider discrete event behaviour, and thus map each allowable input sequence of events to a sequence of output events. It was described by the 9-tuple $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$. Coupled DEVS models consider structure and routing of the atomic DEVS models it contains. It was described by the 6-tuple $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$. For each aspect of the DEVS specification, we have given justification as to why it is necessary, provided the formal specification, and made the link to an implementation in PythonPDEVS.

ACKNOWLEDGMENTS

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

REFERENCES

Chow, A. C. H., and B. P. Zeigler. 1994. "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism". In *Proceedings of the 1994 Winter Simulation Multiconference*, pp. 716–722.

Chow, A. C. H., B. P. Zeigler, and D. H. Kim. 1994. "Abstract simulator for the parallel DEVS formalism". In *AI, Simulation, and Planning in High Autonomy Systems*, pp. 157–163.

Van Tendeloo, Y., and H. Vangheluwe. 2014. "The Modular Architecture of the Python(P)DEVS Simulation Kernel". In *Proceedings of the 2014 Spring Simulation Multiconference*, pp. 387–392.

Van Tendeloo, Y., and H. Vangheluwe. 2016. "An Overview of PythonPDEVS". In *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, edited by C. W. RED, pp. 59–66.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "An evaluation of DEVS simulation tools". *SIMULATION* vol. 93 (2), pp. 103–121.

Van Tendeloo, Y., and H. Vangheluwe. 2018. "Extending the DEVS Formalism with Initialization Information". *ArXiv e-prints*.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation*. 1st ed. Wiley Interscience.

Zeigler, B. P. 1984. *Multifacetted Modelling and Discrete Event Simulation*. 1st ed. Academic 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 is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). He is a member of 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, grafted on the Python programming language. The topic of his PhD is the conceptualization and development of a new (meta-)modelling framework and model management system called the Modelverse.

HANS VANGHELUWE is a Professor in the department of Mathematics and Computer Science at the University of Antwerp (Belgium) and an Adjunct Professor in the School of Computer Science at McGill University (Canada). He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributer to the DEVS community of fundamental and technical research results.