Universiteit Antwerpen

This item is the archived preprint of:

Extending the DEVS formalism with initialization information

Reference:
Van Tendeloo Yentl, Vangheluwe Hans.- Extending the DEVS formalism with initialization information
Arxiv, 2018, 11 p.

uantwerpen.be
Institutional repository IRUA

Extending the DEVS Formalism with Initialization Information

Yentl Van Tendeloo* Hans Vangheluwe*†‡
{Yentl.VanTendeloo,Hans.Vangheluwe}@uantwerpen.be

arXiv:1802.04527v1 [cs.SE] 13 Feb 2018

DEVS is a popular formalism to model system behaviour using a discrete-event abstraction. The main advantages of DEVS are its rigourous and precise specification, as well as its support for modular, hierarchical construction of models. DEVS frequently serves as a simulation "assembly language" to which models in other formalisms are translated, either giving meaning to new (domain-specific) languages, or reproducing semantics of existing languages. Despite this rigourous definition of its syntax and semantics, initialization of DEVS models is left unspecified in both the Classic and Parallel DEVS formalism definition. In this paper, we extend the DEVS formalism by including an initial total state. Extensions to syntax as well as denotational (closure under coupling) and operational semantics (abstract simulator) are presented. The extension is applicable to both main variants of the DEVS formalism. Our extension is such that it adds to, but does not alter the original specification. All changes are illustrated by means of a traffic light example.

Keywords: Classic DEVS, Parallel DEVS, Experimentation, Initialization

1 Introduction

DEVS [19] is a popular formalism to model system behaviour using a discrete-event abstraction. With this abstraction, only a finite number of pertinent events can occur during any bounded time interval. In reaction to events, the model's state variable values change instantaneously. The state remains unaltered between event occurrences. The main advantages of DEVS are its rigourous and precise specification, as well as its support for modular, hierarchical model construction. DEVS frequently serves as a simulation "assembly language" to which models in other formalisms are translated, either giving meaning to new (domain-specific) languages, or reproducing semantics of existing languages [17]. Models in different formalisms can hence be meaningfully combined by mapping them onto DEVS.

Despite the rigour of its syntax and semantics, the initialization of models is unspecified in both Classic [19] and Parallel DEVS [3, 4]. Initialization is important for several reasons. First, it allows for unaltered reuse of models, for example from a model library. A user should not have to know about, let alone modify, the various internal states and configurations of a reused model. Second, initialization is required when restarting a simulation run after it was interrupted, e.g., for fault tolerance reasons. The state has to be re-initialized to the last know state, from which simulation can subsequently resume as if it was never interrupted. Third, initialization is required for dynamic structure and hybrid systems, where the simulation is interrupted due to some system condition, the model is altered, re-initialized to a consistent state, and the simulation resumed.

As it is not part of the specification of the DEVS formalism, different implementations have widely varying ways of supporting initialization, impeding model reuse across simulator implementations. Additionally, realistic initial conditions can often not be expressed without adding artificial initialization behaviour to a model. In this paper, a traffic light example is used to illustrate the need for adding initialization to the DEVS formalism. The effects of the extension on both denotational (closure under coupling, or flattening) and operational semantics (the abstract simulator) are described and demonstrated using the running example. Note that this extension only specifies initialization, leaving all other aspects of DEVS untouched. Our extension is implemented in the PythonPDEVS [14] simulation tool, which is capable of simulating both Classic and Parallel DEVS. As the addition to both is similar, only Classic DEVS is described in detail in this paper.

The remainder of this section presents our motivating example at a conceptual level. Section 2 briefly recaps the Classic DEVS formalisms, without initialization, and models the example in it. Section 3 presents our

*University of Antwerp, Belgium
†Flanders Make, Belgium
‡McGill University, Montréal, Canada

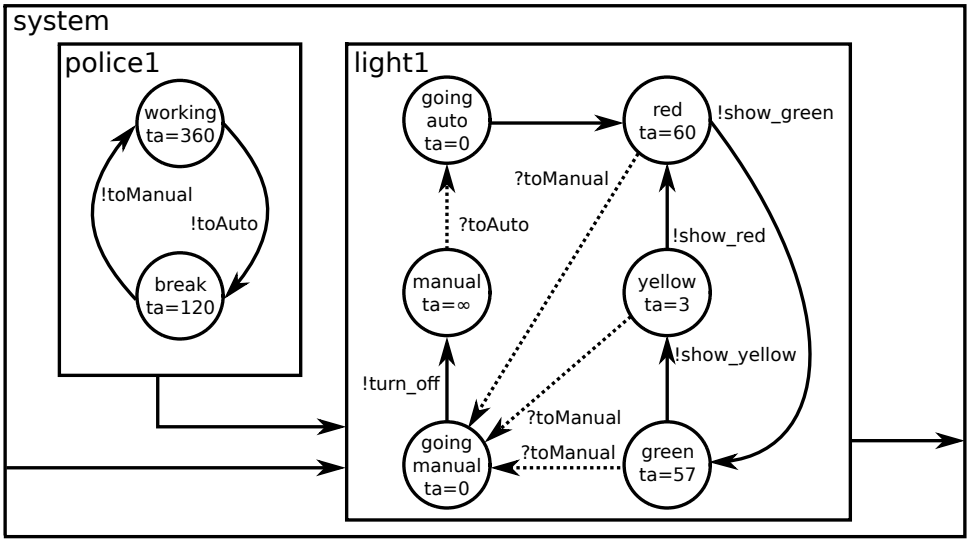

Figure 1: Visual representation of the traffic light model.

extension to the formalism, the initial total state, and describes its influence on the specification and both of its semantics definitions. Changes are demonstrated on the running example. Section 4 presents related work, mainly referring to current methods of initialization and tool support. Section 5 concludes the paper.

1.1 Motivating Example

To illustrate the lack of initialization, and the ensuing problems, a minimal example is used throughout this paper. This running example consists of a simple traffic light which can run autonomously (i.e., alternate between red, yellow, and green) or be interrupted (i.e., be turned off). This traffic light interacts with a policeman, who alternates between "working" (while the traffic light is turned off) and "taking break" (while the traffic light is working autonomously). For the sake of readability of the behaviour traces, artificifial timings were chosen in the model shown in Figure 1. A full Classic DEVS specification is given in Section 2.

From the visual representation of the model, there is no indication in which state a simulation starts. For example, should the traffic light start autonomously or should it be turned off? And what about the policeman? This is the first problem: there is no initial state specified in the DEVS formalism.

Assuming that some initial sequential state is given, a simulation can be performed, leading to the behaviour traces shown in Figure 2[1]. There is however no flexibility in when the policeman can trigger an event. Without altering the delays in the policeman model, it is impossible for the policeman to interrupt the trafficlight at an arbitrary point in time: it must be at time 120 or 360, as these are the time delays in the policaman model. In both cases, however, this coincides with the time at which the light changes from red to green: it is impossible to interrupt at any other time with this model. To allow for arbitrary times, the model needs to be modified, adding artificial states, thereby introducing accidental and unnecessary complexity. This is an indication that the formalism in its current form lacks expressiveness (or in this case, is under-specified). Furthermore, the model remains incompletely initialized for as long as even a single atomic model is still in its initialization state. Ideally, it should be possible to shift behaviour traces of component models independently with respect to one another, such that the same models can be reused unchanged, but that for example the policeman starts in between two state changes, as shown in Figure 3. The initial state of the policeman can thus be set such that the interrupt happens at any desired point in time. This is in essence the elapsed time of the atomic model: how long it has been since the last transition. This is therefore the second problem: the time that was already spent in the initial state at the start of a simulation cannot be set.

[1]For the sake of readability, none of the figures are to scale.

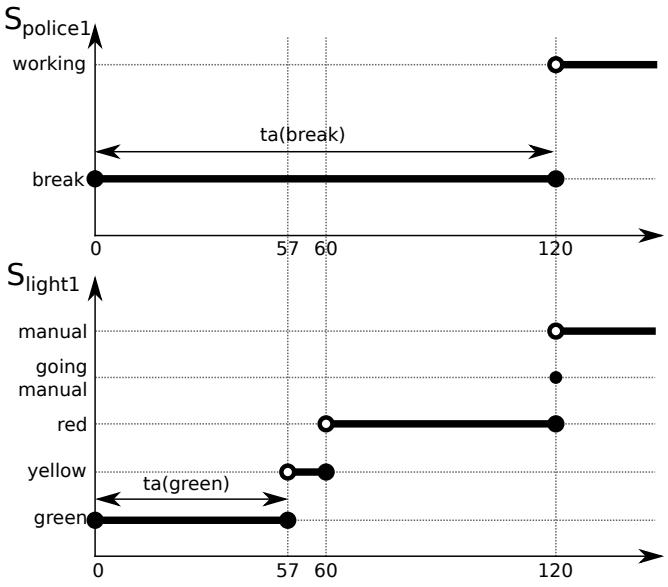

Figure 2: Initialized model without specified initial elapsed times for the component models.

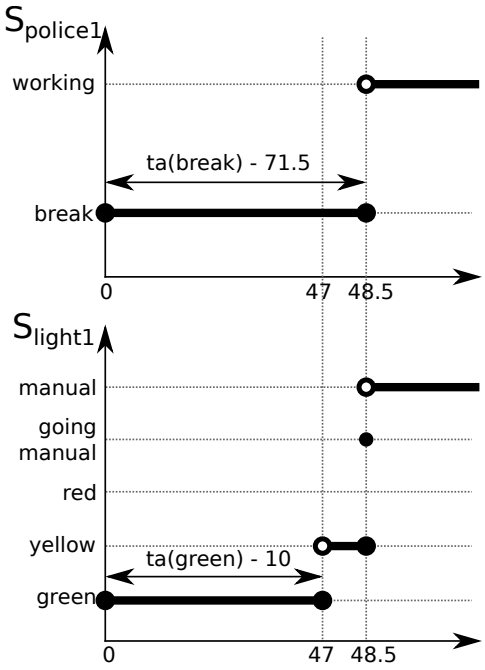

Figure 3: Initialized model with specified initial elapsed times: start of component models' behaviours are shifted in time.

2 Background

In this section, the Classic DEVS formalism is briefly presented to provide the necessary background for our extension to the formalism. The running traffic light example is subsequently specified in DEVS. Only Classic DEVS is presented here as our extension is applicable to both Classic and Parallel DEVS.

For a more detailed explanation of the Classic DEVS formalism, we refer the reader to the main DEVS reference work [19] (including Parallel DEVS) or our Classic DEVS tutorial [15]. Further details of Parallel DEVS can be found in the literature as well [3, 4].

2.1 Classic DEVS

Classic DEVS comprises two types of models: Atomic DEVS models, defining behaviour, and Coupled DEVS models, defining structure.

An Atomic DEVS model is the basic building block. Its structure is shown in Specification 1.

$$AM = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{1}$$

X	*set of input events*
Y	*set of output events*
S	*set of sequential states*
$\delta_{int} : S \to S$	*internal transition function*
$\delta_{ext} : Q \times X \to S$	*external transition function*
$Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)$	*set of total states*
$\lambda : S \to Y \cup \{\phi\}$	*output function, with ϕ the null (nothing happens) event*
$ta : S \to \mathbb{R}^{+}_{0,+\infty}$	*time advance*

Intuitively, the behavioural semantics are as follows. The system enters a sequential state $s \in S$ and schedules an "internal" transition to state $\delta_{int}(s)$ after $ta(s)$. Before undergoing the transition, $\lambda(s)$ is invoked to generate an output event $y \in Y$. If before the scheduled internal transition occurs, an external input event $x \in X$ is received, the scheduled output generation and subsequent transition do not occur. Instead, an "external" transition is made to $\delta_{ext}((s,e),x)$. Here, e is the elapsed time, the time that has passed in the state s since the last transition, until the event was received. No output is generated in this case. Upon arrival in the new state, either through δ_{int} or δ_{ext}, the algorithm repeats.

A Coupled DEVS model is the structuring concept of DEVS, and allows various Atomic and Coupled DEVS models to be combined through parallel composition. Its structure is shown in Specification 2.

$$CM = \langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle \tag{2}$$

X_{self}	*set of input events*
Y_{self}	*set of output events*
D	*set of model instance labels*
$MS = \{M_i \mid i \in D\}$	*set of submodel specifications*
$M_i = \{\langle X_i, Y_i, S_i, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle \mid i \in D\}$	*(atomic) submodel specification*
$IS = \{I_i \mid i \in D \cup \{self\}\}$	*influencee mapping (encoding connection topology)*
$I_i : i \to 2^{D \cup \{self\} \setminus \{i\}}$	*set of influencees' labels of model with label i*
$ZS = \{Z_{i,j} \mid i \in D \cup \{self\}, j \in I_i\}$	*translation mapping*
$Z_{self,j} : X_{self} \to X_j$	*input-to-input translation*
$Z_{i,j} : Y_i \to X_j$	*output-to-input translation*
$Z_{i,self} : Y_i \to Y_{self}$	*output-to-output translation*
$select : 2^D \to D$	*select function*

Intuitively, the semantics are given as follows. The coupled DEVS model instantiates all of its submodels, which are all considered to be atomic DEVS models (as a coupled model can always be flattened to an atomic model), and keeps their references (or labels) in D. The atomic DEVS submodels' specifications are found in MS. Submodels can be connected and the connection topology is encoded in the influencee set IS, which lists for each subcomponent, all the other subcomponents that it influences (i.e., sends its output to). Upon forwarding an event to another subcomponent, the event is translated by ZS, which can map input-to-input (for external input coupling), output-to-input (for internal coupling), and output-to-output (for external output coupling). When multiple internal events are scheduled at the same time, in different sub-models, the *select* function is invoked for tie-breaking.

2.1.1 Running Example

The Classic DEVS specification of the full traffic light model is given below. Note that, as per the DEVS semantics, the output function λ is invoked *before* the internal transition δ_{int} is taken. This explains why the produced events are non-intuitive (e.g., raise *show_yellow* for GREEN). The traffic light atomic DEVS model is shown in Specification 3.

$$Light = \langle X_{light}, Y_{light}, S_{light}, \delta_{int,light}, \delta_{ext,light}, \lambda_{light}, ta_{light} \rangle \tag{3}$$

$$
\begin{aligned}
X_{light} &= \{toAuto, toManual\} \\
Y_{light} &= \{show\_green, show\_yellow, show\_red, turn\_off\} \\
S_{light} &= \{\text{GREEN}, \text{YELLOW}, \text{RED}, \text{GOING\_MANUAL}, \text{GOING\_AUTO}, \text{MANUAL}\} \\
\delta_{int,light} &= \{\text{GREEN} \to \text{YELLOW}, \text{YELLOW} \to \text{RED}, \text{RED} \to \text{GREEN}, \\
&\quad \text{GOING\_MANUAL} \to \text{MANUAL}, \text{GOING\_AUTO} \to \text{RED}\} \\
\delta_{ext,light} &= \{(\text{GREEN}, \_, toManual) \to \text{GOING\_MANUAL}, (\text{YELLOW}, \_, toManual) \to \text{GOING\_MANUAL}, \\
&\quad (\text{RED}, \_, toManual) \to \text{GOING\_MANUAL}, (\text{MANUAL}, \_, toAuto) \to \text{GOING\_AUTO}\} \\
\lambda_{light} &= \{\text{GREEN} \to show\_yellow, \text{YELLOW} \to show\_red, \\
&\quad \text{RED} \to show\_green, \text{GOING\_MANUAL} \to turn\_off, \text{GOING\_AUTO} \to show\_red\} \\
ta_{light} &= \{\text{GREEN} \to 57, \text{YELLOW} \to 3, \text{RED} \to 60, \\
&\quad \text{MANUAL} \to +\infty, \text{GOING\_MANUAL} \to 0, \text{GOING\_AUTO} \to 0\}
\end{aligned}
$$

The policeman's behaviour is simpler and is shown in Specification 4.

$$
Police = \langle X_{police}, Y_{police}, S_{police}, \delta_{int,police}, \delta_{ext,police}, \lambda_{police}, ta_{police} \rangle \tag{4}
$$

$$
\begin{aligned}
X_{police} &= \{\} \\
Y_{police} &= \{toAuto, toManual\} \\
S_{police} &= \{\text{BREAK}, \text{WORKING}\} \\
\delta_{int,police} &= \{\text{BREAK} \to \text{WORKING}, \text{WORKING} \to \text{BREAK}\} \\
\delta_{ext,police} &= \{\} \\
\lambda_{police} &= \{\text{BREAK} \to toManual, \text{WORKING} \to toAuto, \\
ta_{police} &= \{\text{BREAK} \to 120, \text{WORKING} \to 360\}
\end{aligned}
$$

Finally, the atomic models are composed in a coupled DEVS model, shown in Specification 5.

$$
System = \langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle \tag{5}
$$

$$
\begin{aligned}
X_{self} &= \{toAuto, toManual\} \\
Y_{self} &= \{show\_green, show\_yellow, show\_red, turn\_off\} \\
D &= \{light1, police1\} \\
MS &= \{M_{light1} = Light, M_{police1} = Police\} \\
IS &= \{light1 \to \{self\}, self \to \{light1\}, police1 \to \{light1\}\} \\
ZS &= \{Z_{self,light1} = \{toAuto \to toAuto, toManual \to toManual\}, \\
&\quad Z_{police1,light1} = \{toAuto \to toAuto, toManual \to toManual\}, \\
&\quad Z_{light1,self} = \{show\_green \to show\_green, show\_yellow \to show\_yellow, \\
&\qquad show\_red \to show\_red, turn\_off \to turn\_off\}\} \\
select &= \{\{light1, police1\} \to police1, \{ligth1\} \to light1, \{police1\} \to police1\}
\end{aligned}
$$

From this complete DEVS specification example, it is clear that something is lacking: nowhere has been specified what is the initial state, and how long the system has already been in that state.

3 Initial Total State

After the above brief recap of the Classic DEVS formalism and the presentation of our running example in this specification, it has become clear that this model cannot be simulated as-is. Indeed, a simulator for this model has

no way of knowing from which total state to start the simulation. To alleviate this problem, we propose to add an *initial total state* to the DEVS specification. While a minimal extension, this has repercussions on both syntax and semantics of the DEVS formalism.

3.1 Atomic DEVS Specification

In the Atomic DEVS specification, the total initial state $q_{init} \in Q$ is added. Q was previously described as being the set of total states $\{(s,e)|s \in S, 0 \leq e \leq ta(s)\}$, as used by the external transition function δ_{ext}. This alters the atomic DEVS specification to the following.

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

The initial total state comprises the initial state s_{init} and the initial elepased time e_{init}.

The initial state $s_{init} \in S$ specifies the system state in which the simulation commences. Its addition is logical, and has up to now been implemented in different simulation tools, as an implicit ad-hoc extension to the formalism. In the case of our running example, we may specify that the traffic light starts in the GREEN state and the policeman in the BREAK state.

The initial elapsed time e_{init} specifies how long the system has been in this state, without a transition being observed. This is a less obvious addition to the initialization phase, and is ignored by several simulation tools. Nonetheless, we argue for its importance in providing flexibility to the DEVS modeller.

If only an s_{init} is present in the specification, but not e_{init}, it is possible to specify GREEN and BREAK as the initial state for the traffic light and policeman, respectively, but one is restricted to their time advances. Indeed, without e_{init}, the initial elapsed time would be implicitly equal to 0, as was shown in Figure 2. This schedules the first internal transition of the policeman at time 120. The traffic light will have internal transitions at times 57 (to YELLOW), 60 (to RED), and 120 (to GREEN). Following this sequence, the policeman will *always* send its interrupt at the exact same point in time, namely when a switch is made from RED to GREEN. Therefore, it is impossible for the modeller to reproduce the real-world scenario where, for example, the policeman interrupts after the light has been in the YELLOW state for 1.5 time units. To do this, the model itself would have to be drastically modified (e.g., adding an artificial "initialization" state to the policeman model). This severely impedes modular re-use of submodels.

What we wish to achieve is shown in Figure 4, which includes the "negative" simulation time (i.e., what hypothetically happened before simulation, given the specified model). While this figure includes the state trace from before the start of the simulation, we can only go back until the last transition, as we have no knowledge about how we ended up in that state (e.g., before time -10 for *police1*). To remain consistent with the DEVS specification, it is required to alter the duration since the last event for each atomic model individually. By doing this, each individual atomic DEVS model can be shifted relatively to all others. For example, Figure 5 presents two different initial elapsed time configurations, with their effect on the simulation. Note that these additions can easily be taken over to Parallel DEVS as well, without any changes.

In our running example, we augment the models with the following initial total states. For the sake of the closure under coupling, we set the initial elapsed time of the traffic light to 10. Therefore, the light will enter YELLOW at time 47. To ensure that the policeman sends the external interrupt after the light has been in state YELLOW for 1.5, the transition has to happen at time 48.5. To achieve this, we compute its desired initial elapsed time as $e_{init,police1} = 120 - 48.5 = 71.5$. Thus, the policeman will have already spent 71.5 time units in BREAK, and will transition after 48.5 time units, at which point the light will have been in YELLOW for 1.5 time units, as desired. Various configurations exist to achieve the same result, such as having the traffic light start at a different state or with a different elapsed time.

$$q_{init,light1} = (\text{GREEN}, 10) \qquad q_{init,police1} = (\text{BREAK}, 71.5)$$

3.2 Closure Under Coupling

The atomic DEVS specification extension has its repercussions on closure under coupling, where several atomic DEVS models are flattened to a single atomic DEVS model. Indeed, the q_{init} of each submodel, has to be conserved in the flattened atomic model. While several approaches exist, we have opted for a non-invasive approach, which only affects q_{init}.

A definition for q_{init} in terms of the $q_{init,i}$ (i being a submodel label) is presented which leaves the closure under coupling untouched. As $q_{init} \in Q$, it consists of two parts: the initial state s_{init} and the initial elapsed time e_{init}.

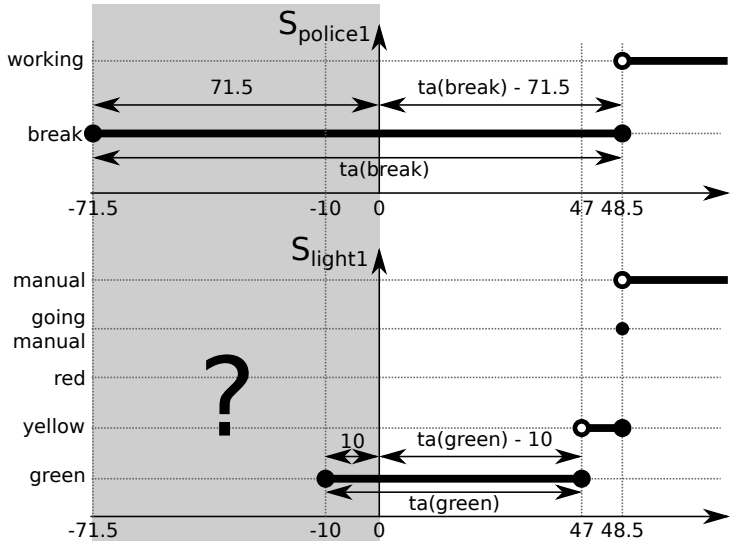

Figure 4: Simulation trace including hypothetical negative simulation time (grayed out).

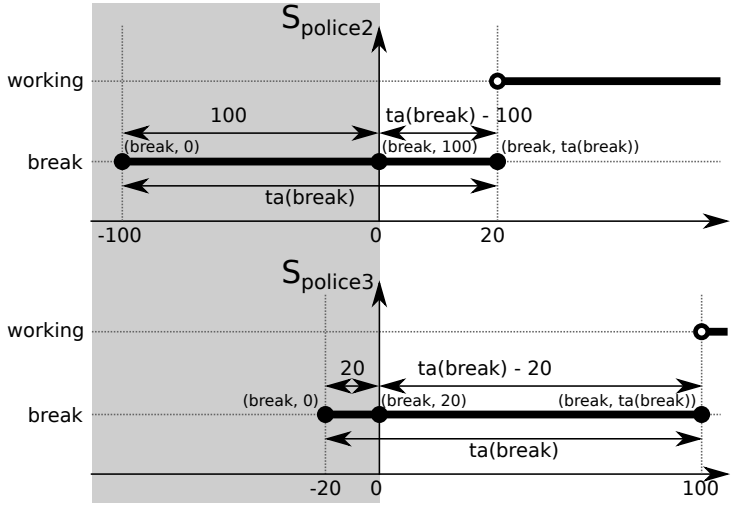

Figure 5: Various options for shifting the police model.

First, we tackle the initial elapsed time e_{init}. As closure under coupling combines all subcomponents, every independent state change of a subcomponent results in a state change of the flattened model. As a logical consequence, the initial elapsed time of the overall system is the time since the last transition of all its subcomponents, i.e., the minimum of all the subcomponents' elapsed times.

$$e_{init} = min_{i \in D}\{e_{init,i}\}$$

Second, we tackle the initial state s_{init}. As mentioned earlier, closure under coupling combines the states of the various subcomponents into a single state. To capture the individual elapsed times, required for the δ_{ext} and δ_{int} functions, the flattened atomic DEVS model not only encodes the combination of all subcomponents' states, but includes for each subcomponent, the *current elapsed time*: S is defined as $\times_{i \in D} Q_i$. Intuitively, one would define s_{init} as the combination of the $q_{init,i}$ values of all the subcomponents, i.e., $s_{init} = \times_{i \in D} q_{init}$, but this is incorrect. Due to the definition of e_{init}, the minimal elapsed time is already taken into account, as shown in Figure 6. Indeed, it has been e_{init} time units since the last transition (to s_{init}), and therefore s_{init} should not contain the elapsed times at initialization, but rather those upon reaching the initial state (i.e., e_{init} time units ago). This can be done by subtracting from each $e_{init,i}$ in s_{init}, the globally minimum value e_{init}. Therefore, we define s_{init} as follows.

$$s_{init} = (..., (s_{init,i}, e_{init,i} - e_{init}), ...)$$

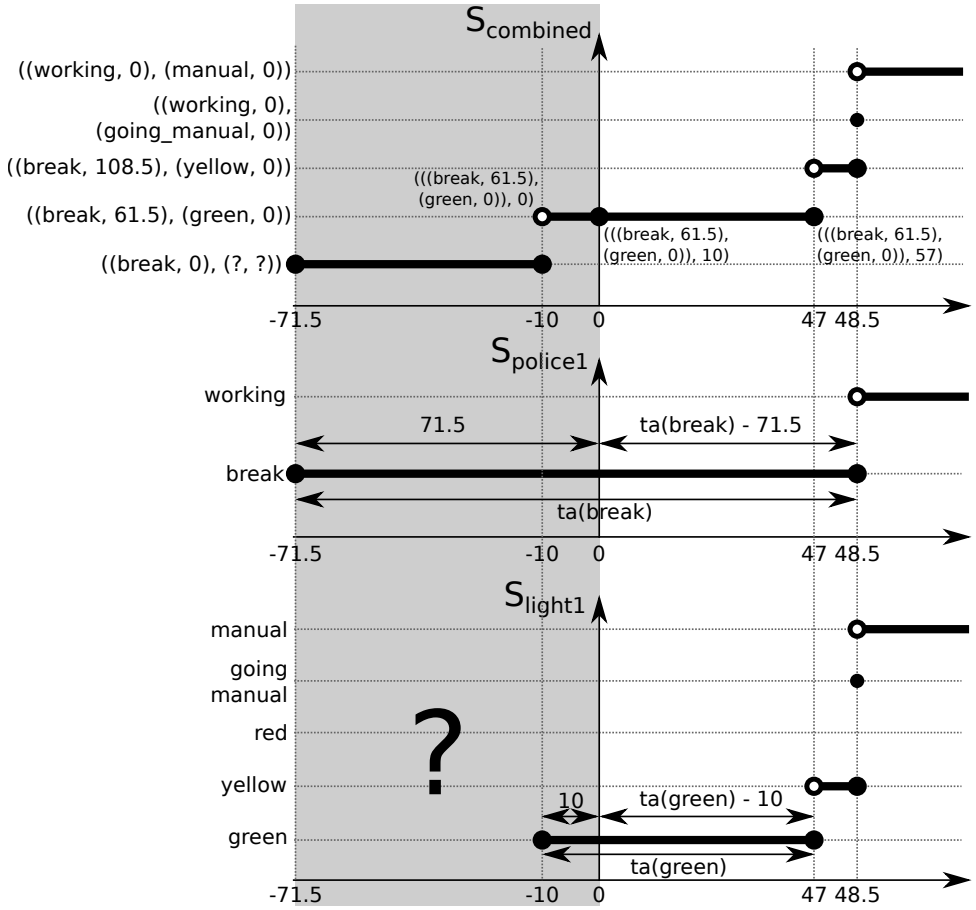

Figure 6: q_{init} in the flattened model.

An alternative would be to set e_{init} to 0 and define s_{init} as the combination of all $q_{init,i}$ values, but this is less intuitive for two reasons. First, the combination of $q_{init,i}$ is never actually "materialized" in simulators, as discrete event simulators jump from one transition time to the next, ignoring all intermediate times (this is the essence of discrete-event simulation). In many cases, no transition happens at simulation time zero, and therefore the state would never have materialized anyway. The option of altering this behaviour for the initialization seems non-intuitive. Second, to keep the definition of elapsed time consistent with that used by the simulator "in regime operation", it must be the time since the last transition of the atomic DEVS model. Setting it to 0 it implies that the system made a transition upon initialization, which is incorrect, unless of course in some rare cases where this occurs in the system being modelled.

For our running example, using the same parameters as before, this results in the following initial total state of the flattened atomic DEVS model.

$$q_{init} = (((\text{GREEN}, 0), (\text{BREAK}, 61.5)), 10)$$

Note that these additions can easily be taken over to Parallel DEVS as well, without any changes.

3.3 Abstract Simulator

The abstract simulator is simpler to alter, as there is already a notion of *initialization message*, often termed *i*. At the start of the simulation, the root coordinator first sends out the message $(i, 0.0)$. Each coordinator or simulator responds to this message with $(done, tn)$, where tn represents the earliest internal event it has scheduled. As in our extended DEVS specification, the atomic models already have the necessary initialization information stored locally, they can set the state and elapsed time upon receiving the *i* message. Depending on these newly set values, the initial $ta(s)$ is computed, which is then sent to the parent simulator. No further changes are required.

4 Related Work

The initialization of DEVS models has up to now been done either by the experimental frame [19, 12], or hard coded in tools.

In the experimental frame approach, the initial total state can be considered as a type of input. The initial total state is thus defined in the experimental frame, and is merely passed along to the DEVS model during initialization. While this approach allows the environment to configure the model as desired, thereby offering flexibility, it is not formalized in the DEVS specification. Indeed, the DEVS semantics, whether it be denotationally using closure under coupling or operationally using the abstract simulator, don't mention anything related to the initial total state. Most of the time, it is a logical decision to have the model itself define what is a consistent state for initialization.

Initialization is often left to tools, which implement this independent of the formalim. We consider several popular DEVS simulation tools, discussed in an earlier survey [16].

Adevs [8] considers all attributes of an atomic model to be part of the state. Therefore, the constructor is responsible for setting all attributes. The initial elapsed time cannot be set.

CD++ [18] similarly considers all attributes to be part of the state, but provides a specific function called `initFunction`. This function is called upon initialization and sets the initial state and the remaining time in this state (σ). While this is functionally equivalent to the elapsed time, as $e = ta(s) - \sigma$, it diverges from the DEVS specification, as it is possible to be in an initial state which would have ordinarily been impossible to reach (e.g., $q_{init} = (\text{YELLOW}, 100)$ for our running example). Indeed, it might be that $\sigma > ta(s)$, which will remain undetected. In contrast, if the elapsed time is given, the definition of Q ensures that $0 \leq e \leq ta(s)$, rendering such inconsistencies impossible.

DEVS-Suite [6] similarly has an `initialize` function which initializes the state and sets the first timeout to use, similar to CD++. As its approach is the same as that of CD++, it shares the same problems.

MS4Me [11] again uses the same approach, but requires the syntax `to start hold in S_INIT for time REMAINING_INIT!` to achieve the same result. Again, the same problems occur.

PowerDEVS [2] has the same approach, where the function is called `init`. Again, the function initializes the state and returns the remaining time until the first internal transition.

PythonPDEVS [13], our Classic and Parallel DEVS simulator, supports the definition of an initial total state as presented in this paper. Like all tools, it is required to specify the initial state (assign to `self.state`), but additionally, the elapsed time can be set (assign to `self.elapsed`). By specifying the elapsed time, instead of the remaining time, the timings are guaranteed to be consistent.

VLE [9] uses the constructor to initialize the state, as in adevs, and similarly does not present an option to specify the initial elapsed or remaining time.

X-S-Y [5] assigns the initial state to the `self.phase` attribute, but does not allow for either the elapsed or remaining time to be set.

While several tools have addressed the missing initialization in the DEVS specification, they do so in widely varying ways. Syntactical differences are found among all tools, as there is no agreed upon name or method of adding initialization information. More importantly, semantical differences exist as well: tools such as adevs, vle, and X-S-Y can not specify the initial time, and are therefore restricted in their (model reuse) flexibility. Other tools, such as CD++, DEVS-Suite, MS4Me, and PowerDEVS, allow the remaining time to be specified, though this potentially results in inconsistent situations, as there is no formal constraint on this value. Finally, PythonPDEVS implements the initial total state using the elapsed time, thereby performing consistency checking: the elapsed time e is immediately compared to the time advance ta. In the light of DEVS standardization efforts [10, 1], it is problematic that various tools implement different methods of initialization. On this topic, a "DEVS compliance checklist" was previously introduced [7], and later slightly extended [16], to which we believe the initial total state should now be added.

5 Conclusion

DEVS has since long been acclaimed for its rigourous and precise specification for models with a discrete event abstraction. Despite its rigour being one of its main advantages, model initialization is left unspecified, leading to various implementations. This paper has presented an addition to the DEVS specification, the initial total state q_{init}, which formalizes model initialization. The influence of this addition on the specification and closure under coupling was presented in detail, both at the level of the specification, and using a traffic light example. Classic DEVS was used to present our contribution, though the addition can be applied to Parallel DEVS as well.

References

[1] Khaldoon Al-Zoubi and Gabriel Wainer. Interfacing and coordination for a DEVS simulation protocol standard. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 300–307, 2008.

[2] Federico Bergero and Ernesto Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87:113–132, 2011.

[3] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Multiconference*, pages 716–722, 1994.

[4] Alex Chung Hen Chow, Bernard P. Zeigler, and Doo Hwan Kim. Abstract simulator for the parallel DEVS formalism. In *AI, Simulation, and Planning in High Autonomy Systems*, pages 157–163, 1994.

[5] Moon Ho Hwang. X-S-Y. https://code.google.com/p/x-s-y/, 2012.

[6] Sungung Kim, Hessam S. Sarjoughian, and Vignesh Elamvazhuthi. DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 161:1–161:7, 2009.

[7] Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for DEVS formalism implementations. In *Proceedings of the 2011 Spring Simulation Multiconference*, pages 183–188, 2011.

[8] James J. Nutaro. adevs. http://www.ornl.gov/~1qn/adevs/, 2015.

[9] Gauthier Quesnel, Raphaël Duboz, Éric Ramat, and Mamadou K. Traoré. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Multiconference*, pages 367–374, 2007.

[10] Hessam S. Sarjoughian and Yu Chen. Standardizing DEVS models: an endogenous standpoint. In *Proceedings of the 2011 Spring Simulation Multiconference*, pages 266–273, 2011.

[11] Chungman Seo, Bernard P. Zeigler, Robert Coop, and Doohwan Kim. DEVS modeling and simulation methodology with MS4Me software. In *Proceedings of the 2013 Spring Simulation Multiconference*, pages 33:1–33:8, 2013.

[12] Mamadou K. Traoré and Alexandre Muzy. Capturing the dual relationship between simulation models and their context. *Simulation Modelling Practice and Theory*, 14(2):126–142, 2006.

[13] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Spring Simulation Multiconference*, pages 387–392, 2014.

[14] Yentl Van Tendeloo and Hans Vangheluwe. An overview of PythonPDEVS. In Collectif Workshop RED, editor, *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, pages 59–66, 2016.

[15] Yentl Van Tendeloo and Hans Vangheluwe. Classic DEVS modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, pages 644 – 656. IEEE, December 2017.

[16] Yentl Van Tendeloo and Hans Vangheluwe. An evaluation of DEVS simulation tools. *SIMULATION*, 93(2):103–121, 2017.

[17] Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134, 2000.

[18] Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[19] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.

Acknowledgements

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO). This research was also partially supported by Flanders Make vzw, the Flemish strategic research centre for the manufacturing industry. The authors wish to thank Joey De Pauw for pointing out an inconsistency between q_{init} and the specification of δ_{int} in our lecture notes.