DEVS Modelling and Simulation

Yentl Van Tendeloo   Hans Vangheluwe

Presenters

YENTL VAN TENDELOO is a PhD student 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 parallel and distributed simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS with support for computational resource usage models. The topic of his PhD is the conceptualization and development of a new meta-modelling framework and model management system called the Modelverse. His e-mail address is Yentl.VanTendeloo@uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium), an Adjunct Professor at McGill University (Canada) and an Adjunct Professor at the National University of Defense Technology (NUDT) in Changsha, China. 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. In a variety of projects, often with industrial partners, he develops and applies the model-based theory and techniques of Multi-Paradigm Modelling (MPM). His current interests are in domain-specific modelling and simulation, including the development of graphical user interfaces for multiple platforms. His e-mail address is Hans.Vangheluwe@uantwerpen.be.

Abstract

DEVS is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. At this abstraction level, a timed sequence of pertinent "events" input to a system (or internal, in the case of timeouts) cause instantaneous changes to the state of the system. Due to its rigorous formal definition, and its support for modular composition, several advantages are achieved: 1) it is an appropriate formalisms to model (software) architectures with a precise behavioural description, granting performance analysis, real-time execution, and interoperation with actual systems; 2) it is a "simulation assembly language" to which other simulation languages can be mapped, granting formalism coupling at the DEVS level; and 3) it is a hierarchical framework for co-simulation or orchestration of discrete-event simulators. 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 (behavioural) models are introduced first, after which Coupled (structural) models are introduced. We continue to actual applications of DEVS, for example in performance analysis of queueing systems. All examples are presented with the tool PythonPDEVS, though this introduction is equally applicable to other DEVS tools.

Keywords

Simulation, discrete event, semantic domain, performance analysis, orchestration, DEVS

Length

The proposed length of the tutorial is three hours.

Level of the Tutorial

Introductory. No knowledge of simulation or DEVS is required.

Target Audience

Modellers with an interest in (discrete-event) simulation and performance analysis.

Novelty

This tutorial has previously been given at other conferences.

At the Spring Simulation Multi-Conference (SpringSim)[1], this tutorial was given in 2016 and 2017. Due to the success of this tutorial, an extended tutorial (adding an advanced part) is scheduled for the 2018 edition of this conference. The SpringSim version of the tutorial handled a minor variation of the formalism, which was more appropriate to the target audience (Parallel DEVS instead of Classic DEVS). Additionally, a different viewpoint was taken to motivate the tutorial, as most attendees of this conference are already somewhat familiar with the DEVS formalism and its simulators. For MoDELS, a more elaborate motivation will be given for the use of DEVS in the context of Model-Driven Engineering, as well as more focus on its place in the modelling process.

At the Winter Simulation Conference (WSC)[2], this tutorial was given in 2017. This version of the tutorial was again targeted towards simulation practitioners, which will be different for the MoDELS tutorial.

[1]http://scs.org/springsim

[2]http://meetings2.informs.org/wordpress/wsc2017/

Description of the Tutorial and Intended Outline

- Introduction to Modelling and Simulation, with a focus on different abstractions (e.g., discrete event, discrete time), different modes of execution (e.g., as fast as possible, realtime).
- Motivate the use of DEVS as a discrete event formalism. In the context of MoDELS, three viewpoints are taken to highlight its potential use:

    1) DEVS is an appropriate formalism to model (software) architectures with a precise behavioural description. Thanks to DEVS, we achieve several benefits. First, performance analysis becomes possible with regard to constrained resources (e.g., time or memory). This is often faster than actual execution, is guaranteed to be deterministic, and allows for what-if analysis. Second, the model can be simulated in real-time, thereby effectively executing the real application. Third, thanks to its modularity, it is possible to transparently replace parts of the simulation with the actual system that was being modeled. As the same model can be used for these three applications, this increases consistency.
    2) DEVS can be used as a semantic domain for the meaningful combination of different formalisms if the goal is simulation. DEVS has been identified as a "simulation assembly language" to which other simulation languages can be mapped. Thanks to the modularity of DEVS, these DEVS models originating from different domain-specific models can then be coupled at the level of DEVS.
    3) DEVS can be used as a hierarchical framework for co-simulation or orchestration of discrete-event simulators.

- Introduction to DEVS as a discrete event formalism and our simulator. While our tutorial uses PythonPDEVS, all concepts are tool-independent and can be applied in different DEVS modelling and simulation tools.
- Introduction of the running example: a simple traffic light that can be interrupted by a policeman. Throughout this tutorial, this model is incrementally constructed, highlighting the different aspects of DEVS.
- Atomic DEVS models are the behavioural atomic blocks of a DEVS model. We start from an autonomous traffic light, which we subsequently extend with input/output events, and later with external event processing. Each increment to the traffic light example is accompanied by additional elements of the atomic DEVS specification, an intuitive description of the semantics, and a hand-on execution of the model using PythonPDEVS. The complete formal specification is as follows:

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

X	*set of input events*
Y	*set of output events*
S	*set of sequential states*
$q_{init} \in Q$	*initial total state*
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$	*set of total states*
$\delta_{int} : S \rightarrow S$	*internal transition function*
$\delta_{ext} : Q \times X \rightarrow S$	*external transition function*
$\lambda : S \rightarrow Y \cup \{\phi\}$	*output function*
$ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$	*time advance*

- Coupled DEVS models as the structuring concept of DEVS. We create a traffic light system, which combines both a traffic light and a police, which are coupled together. The various aspects of the coupled DEVS specification are incrementally added, focussing on their importance and the offered flexibility. Again, this is coupled with a hands-on application using PythonPDEVS. The complete specification of a coupled DEVS model is as follows:

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

X_{self}	*set of input events*
Y_{self}	*set of output events*
D	*set of model instance labels*
$MS = \{AM_i \mid i \in D\}$	*set of submodels*
$IS = \{I_i \mid i \in D \cup \{self\}\}$	*topology*
$I_i = 2^{D \cup \{self\} \setminus \{i\}}$	*set of influencees' labels*
$ZS = \{Z_{i,j} \mid i \in D \cup \{self\}, j \in I_i\}$	*translation*
$Z_{self,j} : X_{self} \rightarrow X_j$	*input-to-input translation*
$Z_{i,j} : Y_i \rightarrow X_j$	*output-to-input translation*
$Z_{i,self} : Y_i \rightarrow Y_{self}$	*output-to-output translation*
$select : 2^D \rightarrow D$	*select function*

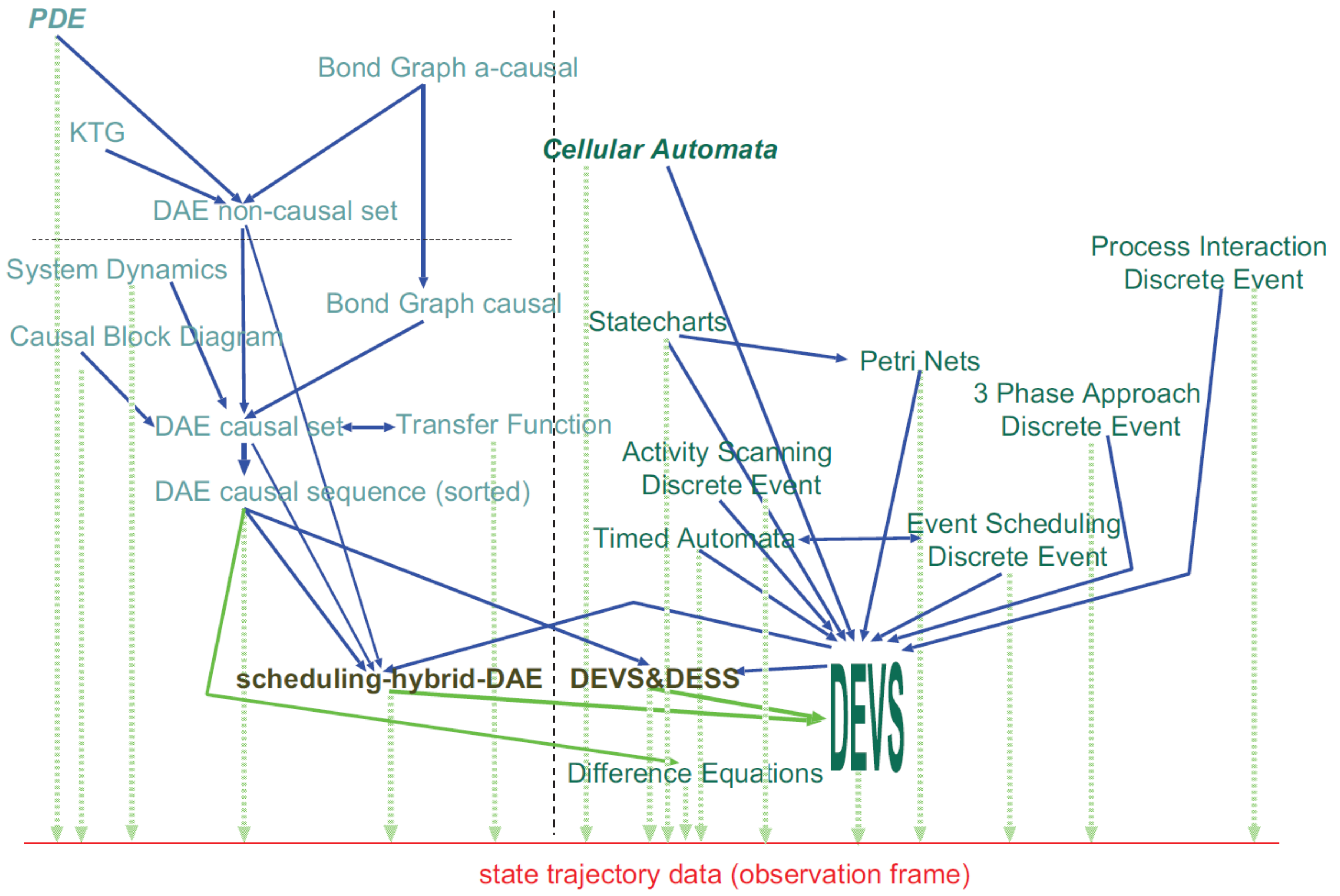
- Semantics of DEVS are described more formally, both operationally (abstract simulator) and denotationally (closure under coupling). This is presented at a high level of abstraction.
- As a more advanced application of DEVS, we present a simple queueing network both conceptually and in PythonPDEVS. Constructing this model, attendees become familiar with common DEVS modelling patterns and learn the caveats of DEVS. This model can be simulated for efficient performance analysis and what-if analysis, teaching attendees how to analyze DEVS simulation results.
- If sufficient time remains, pointers are given on DEVS variants (Parallel DEVS, Dynamic Structure DEVS, Cell DEVS), tool support (adevs, vle, X-S-Y, . . .), and so on.

Required Infrastructure

Only a data projector is required.

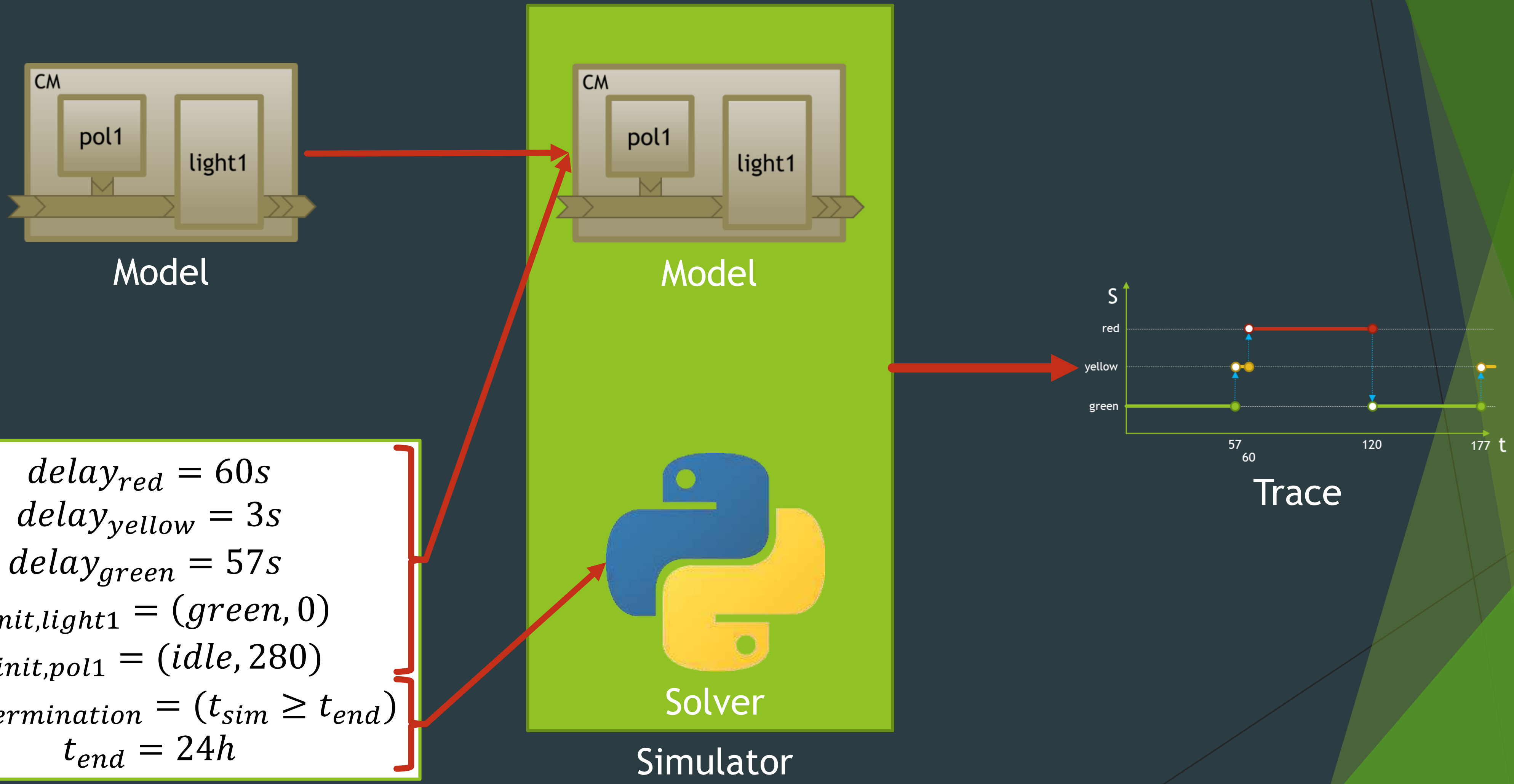
Sample Slides

Some sample slides of the previous versions of the presentation are attached.



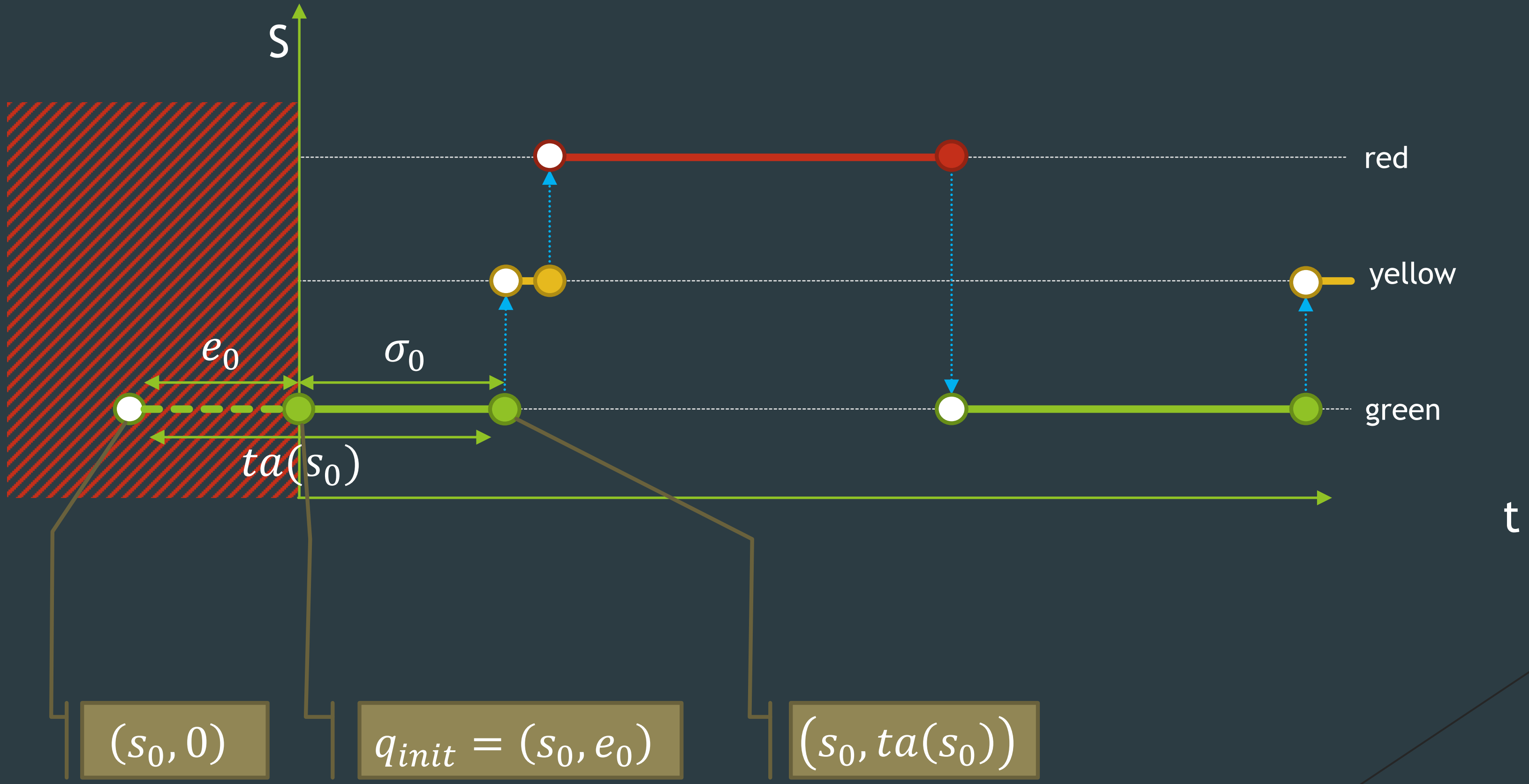
Vangheluwe, Hans. DEVS as a common denominator for multi-formalism hybrid systems modelling.
In proceedings of the International Symposium on Computer-Aided Control System Design, pp. 129-134. 2000.

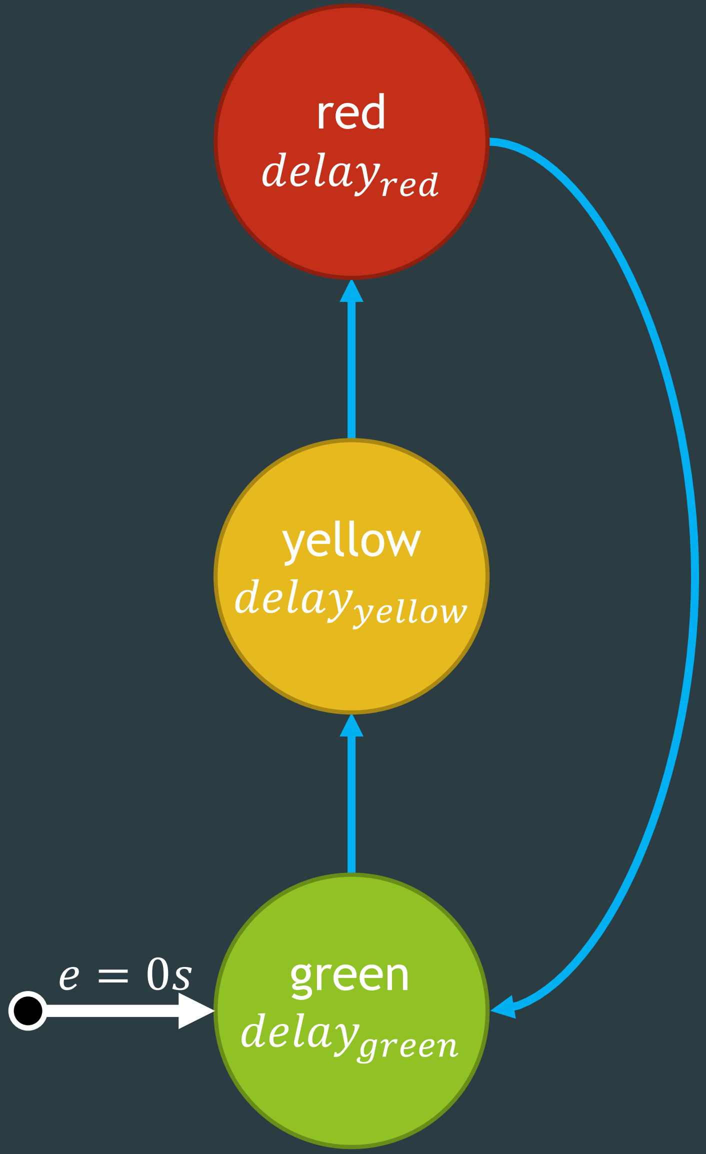
Simulation



$$delay_{red} = 60s$$
$$delay_{yellow} = 3s$$
$$delay_{green} = 57s$$
$$q_{init,light1} = (green, 0)$$
$$q_{init,pol1} = (idle, 280)$$
$$cond_{termination} = (t_{sim} \geq t_{end})$$
$$t_{end} = 24h$$

Initialization of Initial State





Autonomous (no output)

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

S : set of sequential states

S = {red, yellow, green}

$\delta_{int} : S \rightarrow S$

δ_{int} = {red → green,
green → yellow,
yellow → red}

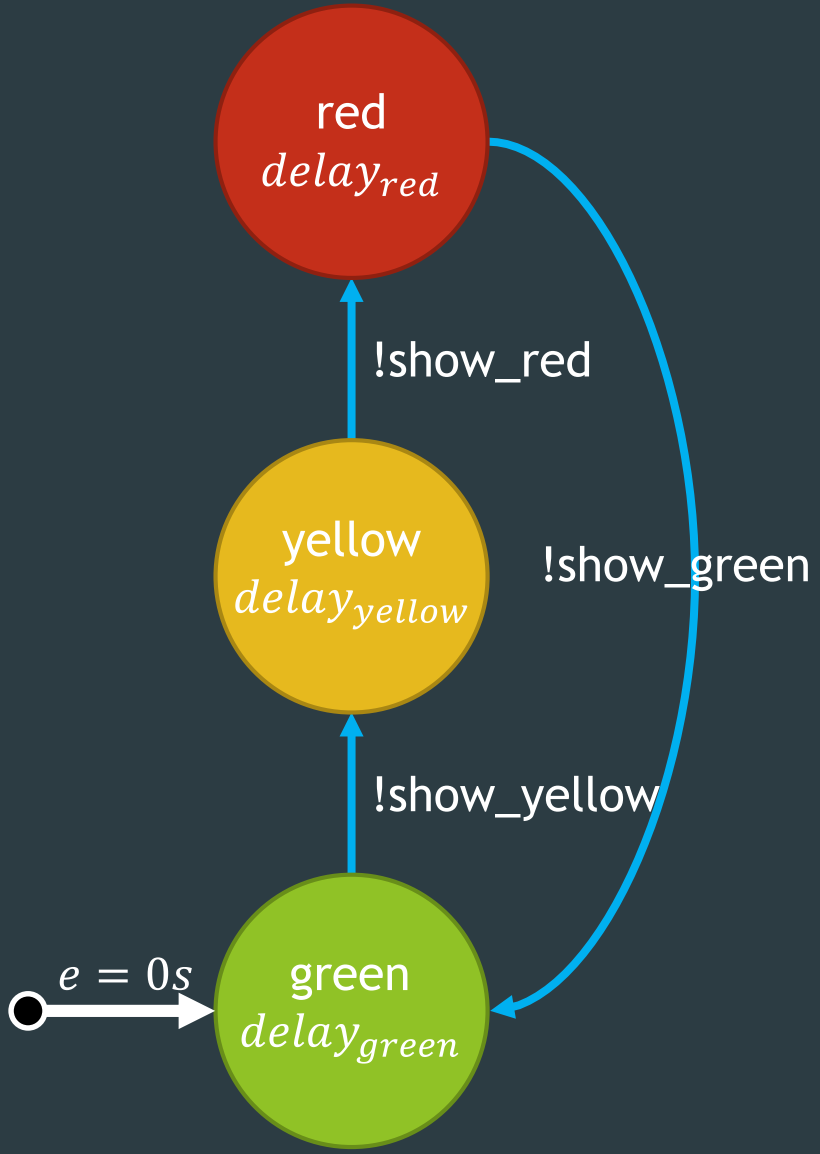
$ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$

ta = {red → $delay_{red}$,
green → $delay_{green}$,
yellow → $delay_{yellow}$}

q_{init} : Q - set of total states

$Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$

q_{init} = (green, 0)



Autonomous (with output)

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

S = {red, yellow, green}
δ_{int} = { red → green,
green → yellow,
yellow → red}
q_{init} = (green, 0)
ta = {red → $delay_{red}$,
green → $delay_{green}$,
yellow → $delay_{yellow}$}

Y : set of output events
Y = {“show_red”, “show_green”, “show_yellow”}

$\lambda : S \to Y \cup \{\phi\}$
λ = { green → “show_yellow”,
yellow → “show_red”,
red → “show_green”}

Abstract Syntax

S = {red, yellow, green}
q_{init} = (green, 0)
δ_{int} = { red → green,
green → yellow,
yellow → red}
ta = {red → $delay_{red}$,
green → $delay_{green}$,
yellow → $delay_{yellow}$}
Y = {“show_red”,
“show_green”,
“show_yellow”}
λ = {green → “show_yellow”,
yellow → “show_red”,
red → “show_green”}

Operational Semantics

```
time = 0
current_state = initial_state
last_time = -initial_elapsed
while not termination_condition():
    time = last_time + ta(current_state)
    output($\lambda$(current_state))
    current_state = $\delta_{int}$(current_state)
    last_time = time
```

Concrete Syntax

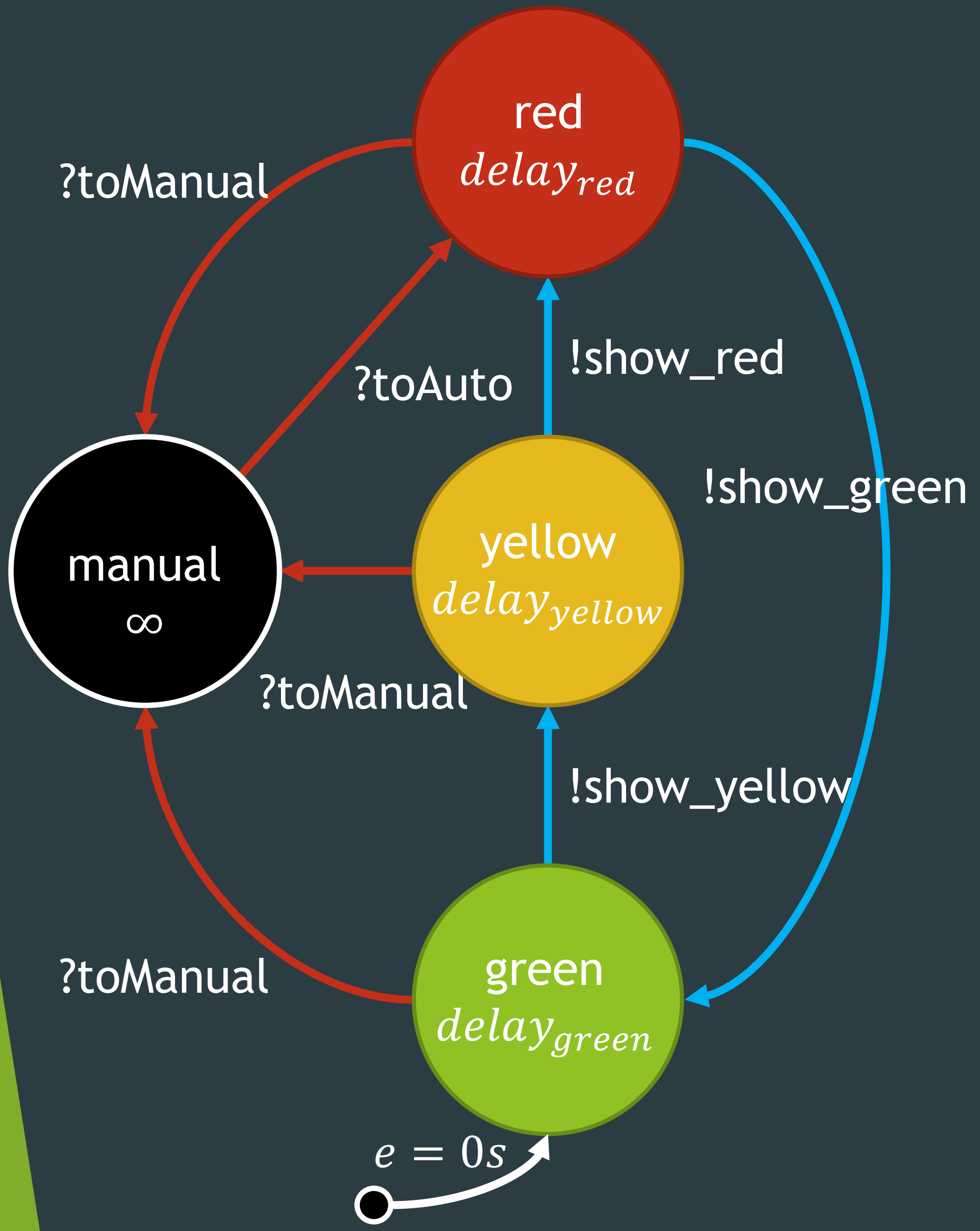
atomic_out.py

```
from pypdevs.DEVS import *

class TrafficLightWithOutput(AtomicDEVS):
    def __init__(self, ...):
        AtomicDEVS.__init__(self, “light”)
        self.observe = self.addOutPort(“observer”)
        ...
    ...

    def outputFnc(self):
        state = self.state
        if state == “red”:
            return {self.observe: “show_green”}
        elif state == “yellow”:
            return {self.observe: “show_red”}
        elif state == “green”:
            return {self.observe: “show_yellow”}
```

Reactive

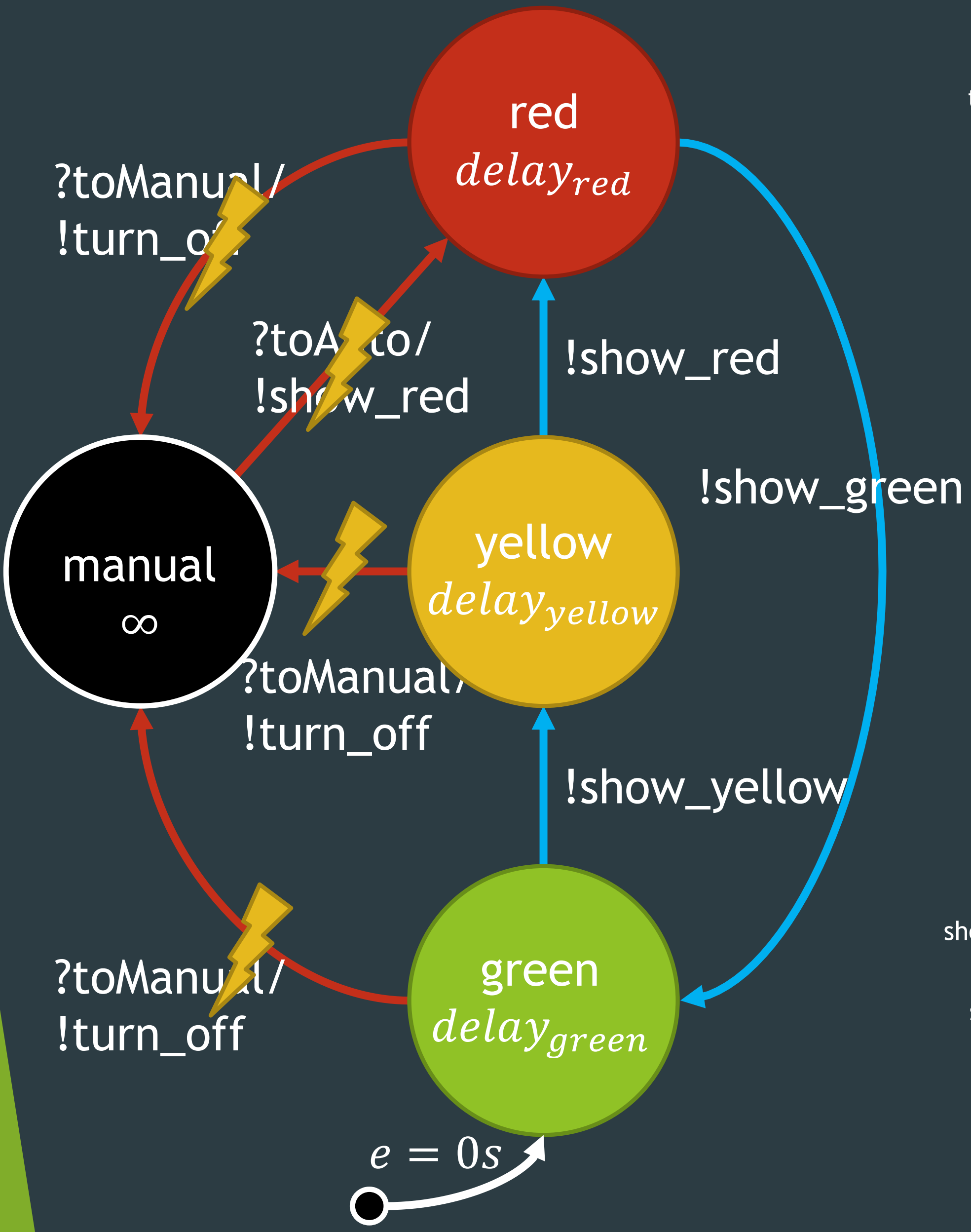


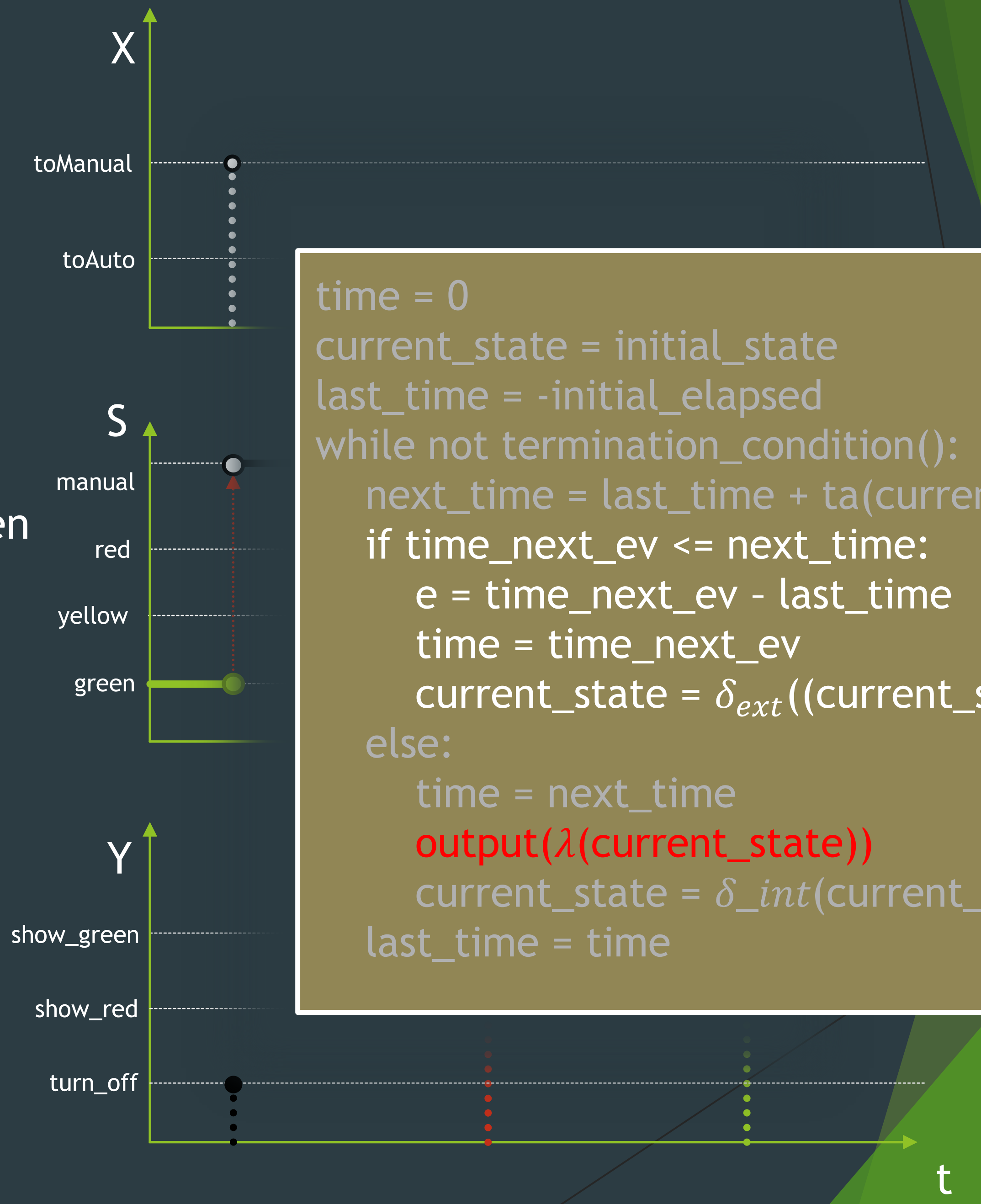
$M = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$

Y = {“show_red”, “show_green”, “show_yellow”}
S = {red, yellow, green, manual}
q_{init} = (green, 0)
δ_{int} = {red → green,
green → yellow,
yellow → red}
λ = {green → “show_yellow”,
yellow → “show_red”,
red → “show_green”}
ta = {red → $delay_{red}$,
green → $delay_{green}$,
yellow → $delay_{yellow}$,
manual → +∞}

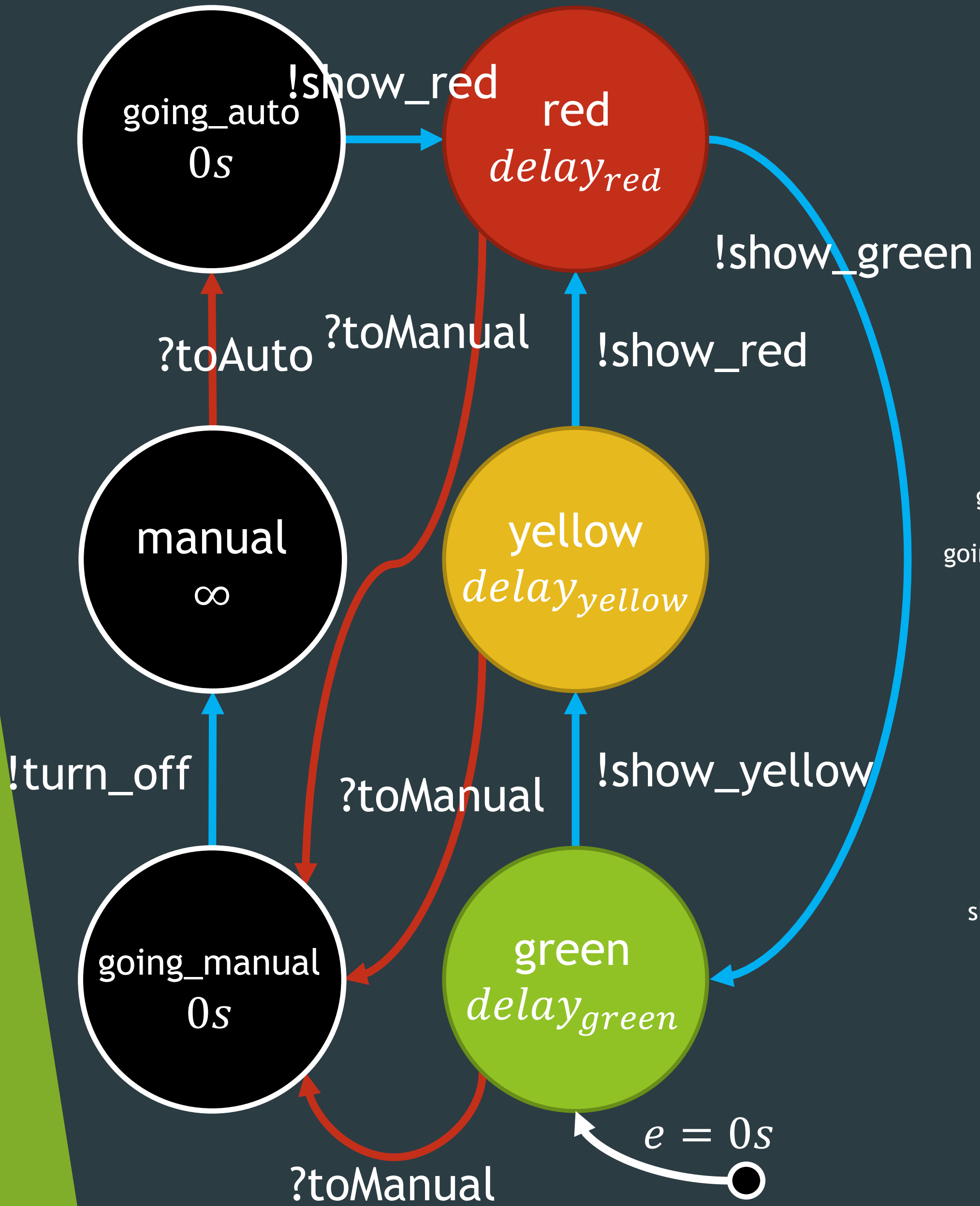
X : set of input events
X = {“toAuto”, “toManual”}

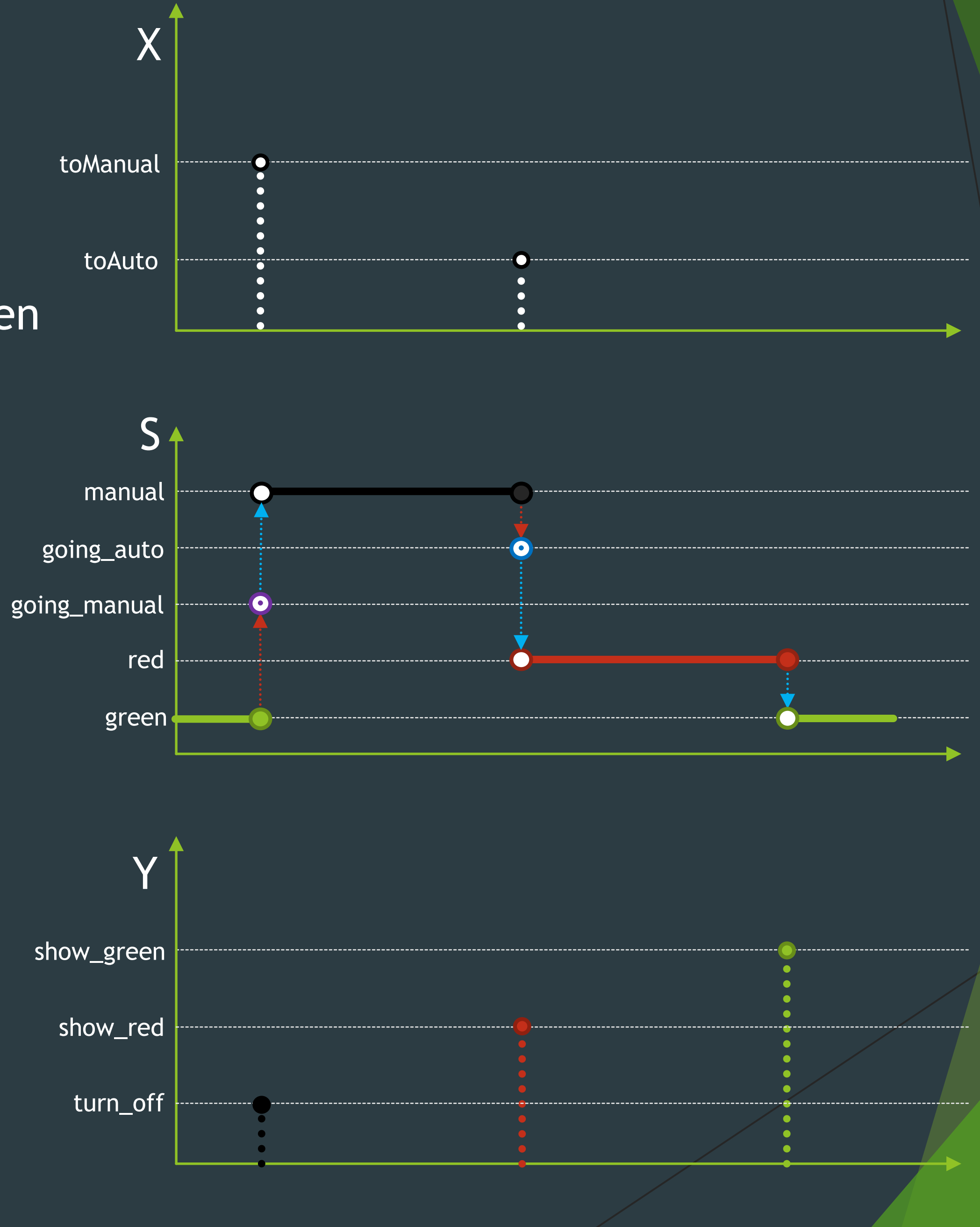
$\delta_{ext} : Q \times X \to S$
$Q = \{(s, e) | s \in S, 0 \le e \le ta(s)\}$
δ_{ext} = {((*, *), “toManual”) → “manual”,
((“manual”, *), “toAuto”) → “red”}

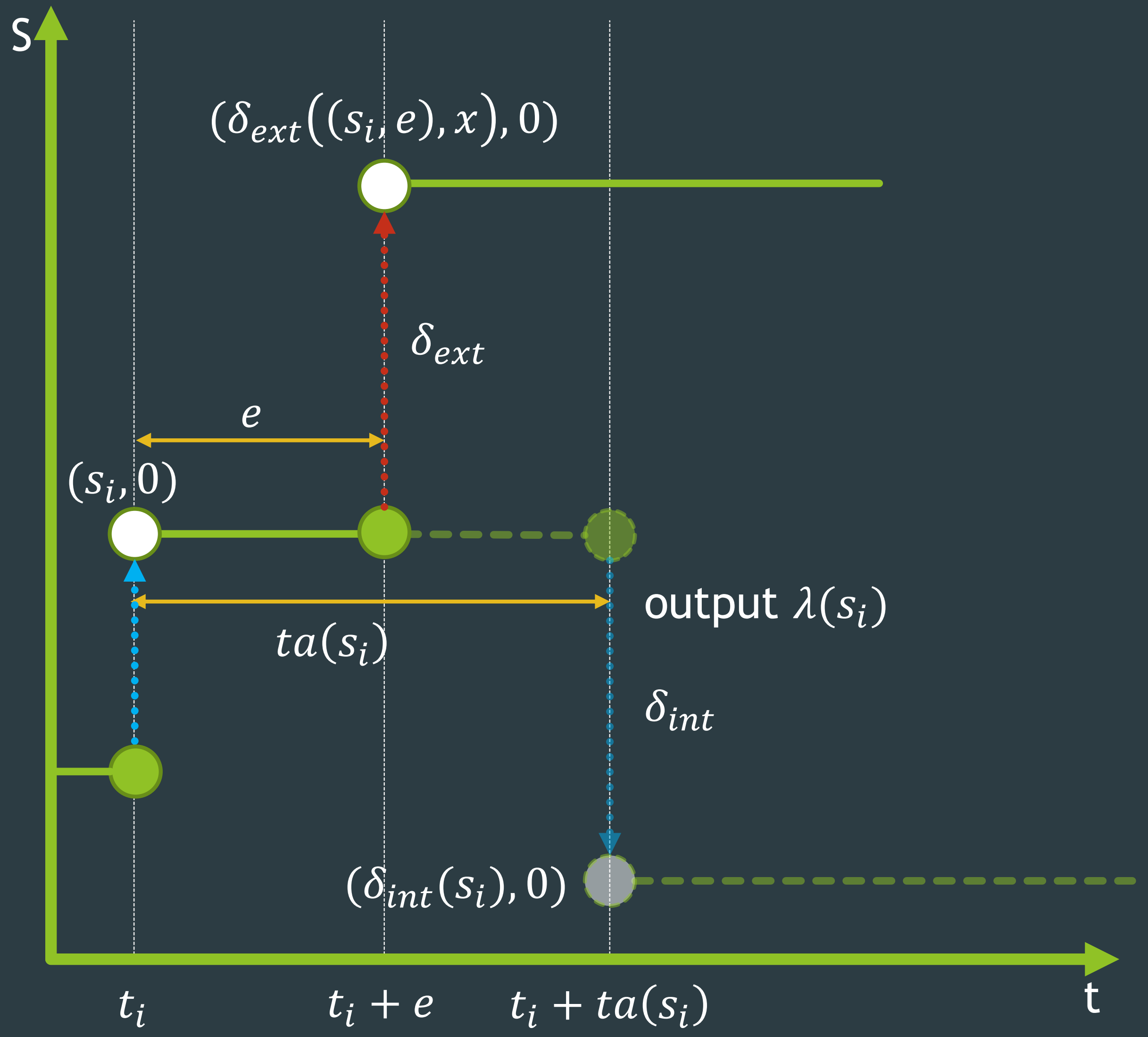




```
time = 0
current_state = initial_state
last_time = -initial_elapsed
while not termination_condition():
    next_time = last_time + ta(current_state)
    if time_next_ev <= next_time:
        e = time_next_ev - last_time
        time = time_next_ev
        current_state = $\delta_{ext}$((current_state, e), next_ev)
    else:
        time = next_time
        output($\lambda$(current_state))
        current_state = $\delta\_int$(current_state)
    last_time = time
```


going_auto
$0s$
red
$delay_{red}$
manual
∞
yellow
$delay_{yellow}$
going_manual
$0s$
green
$delay_{green}$
!show_red
!show_green
?toAuto
?toManual
!show_red
!turn_off
?toManual
!show_yellow
?toManual
$e = 0s$


X
toManual
toAuto
t
S
manual
going_auto
going_manual
red
green
t
Y
show_green
show_red
turn_off
t

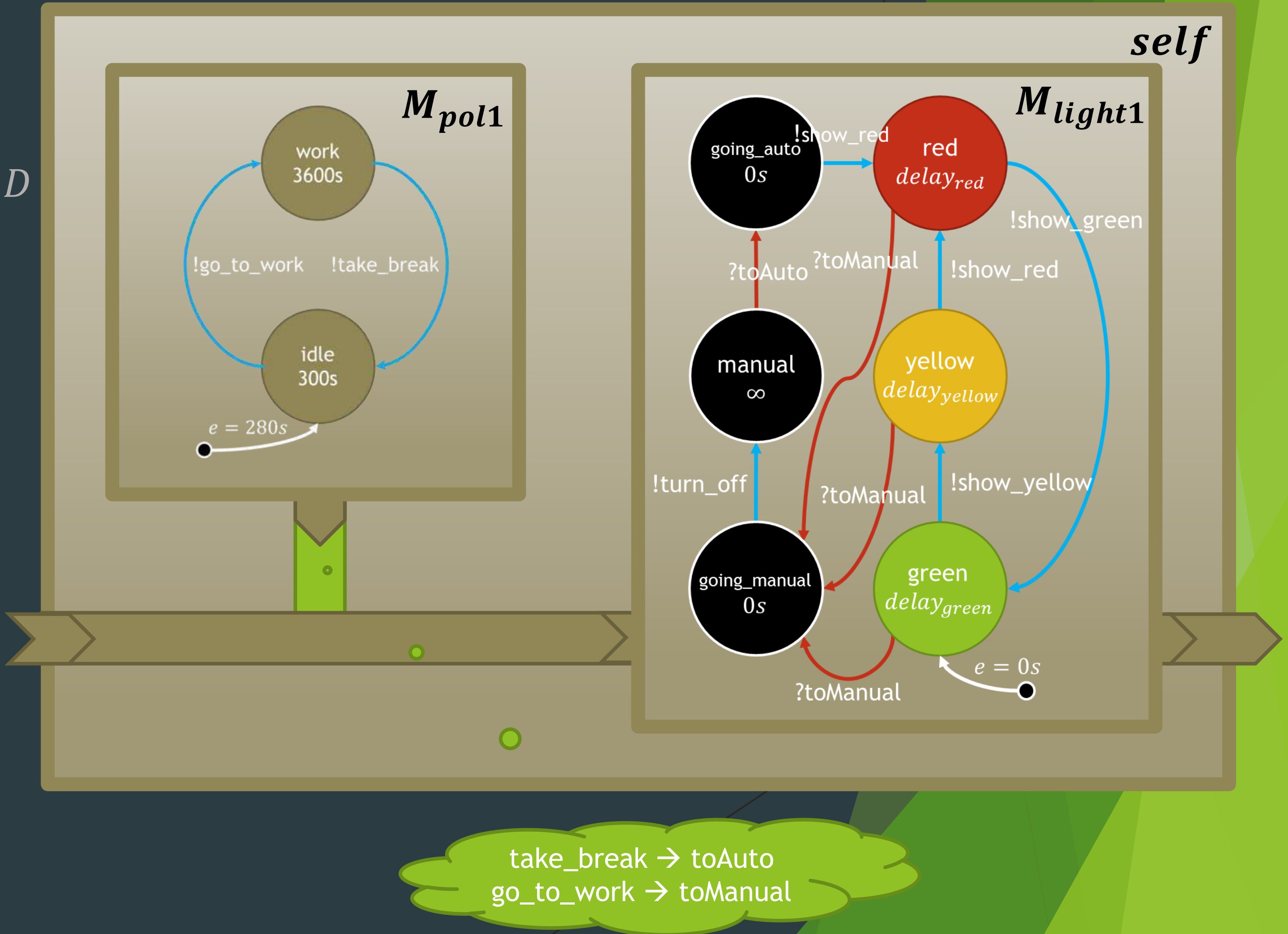

S
$(\delta_{ext}((s_i, e), x), 0)$
δ_{ext}
e
$(s_i, 0)$
output $\lambda(s_i)$
$ta(s_i)$
δ_{int}
$(\delta_{int}(s_i), 0)$
t_i
$t_i + e$
$t_i + ta(s_i)$
t

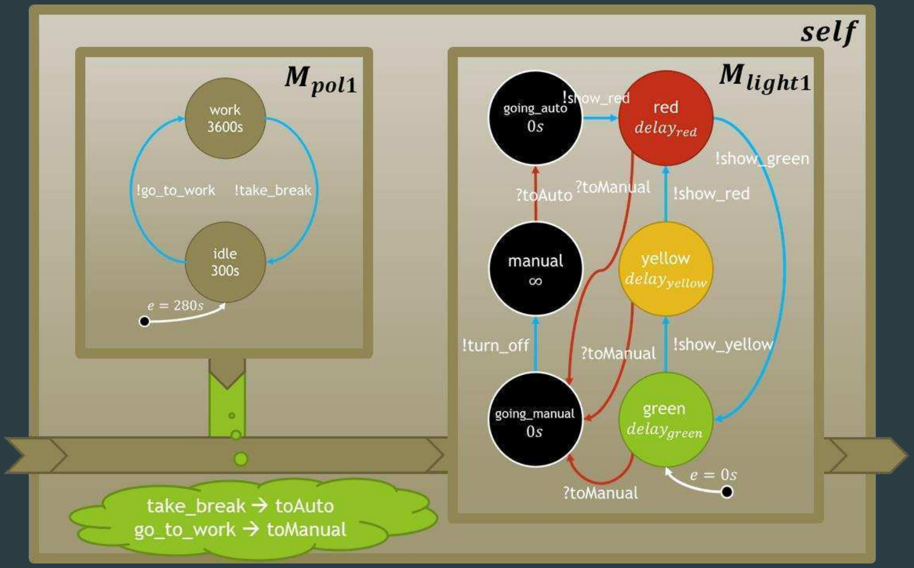
$$C = \langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle$$
$$MS = \{M_i | i \in D\}$$
$$M_i = \langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \lambda_i, ta_i \rangle, \forall\, i \in D$$

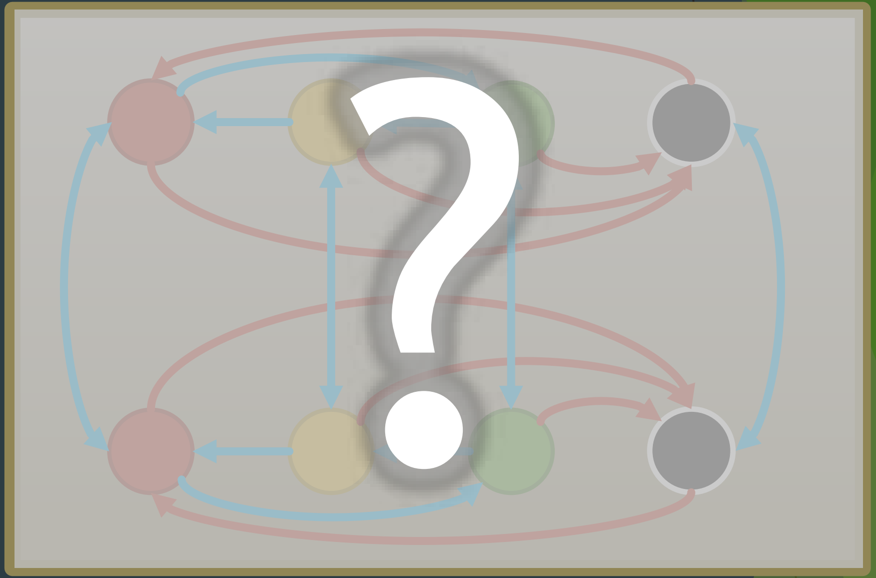
$$IS = \{I_i | i \in D \cup \{self\}\}$$
$$\forall\, i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$$
$$\forall\, i \in D \cup \{self\} : i \notin I_i$$

$$ZS = \{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$$
$$Z_{self,j} : X_{self} \rightarrow X_j, \forall\, j \in D$$
$$Z_{i,self} : Y_i \rightarrow Y_{self}, \forall\, i \in D$$
$$Z_{i,j} : Y_i \rightarrow X_j, \forall\, i, j \in D$$

$$select : 2^D \rightarrow D$$
$$\forall\, E \subseteq D, E \neq \emptyset : select(E) \in E$$

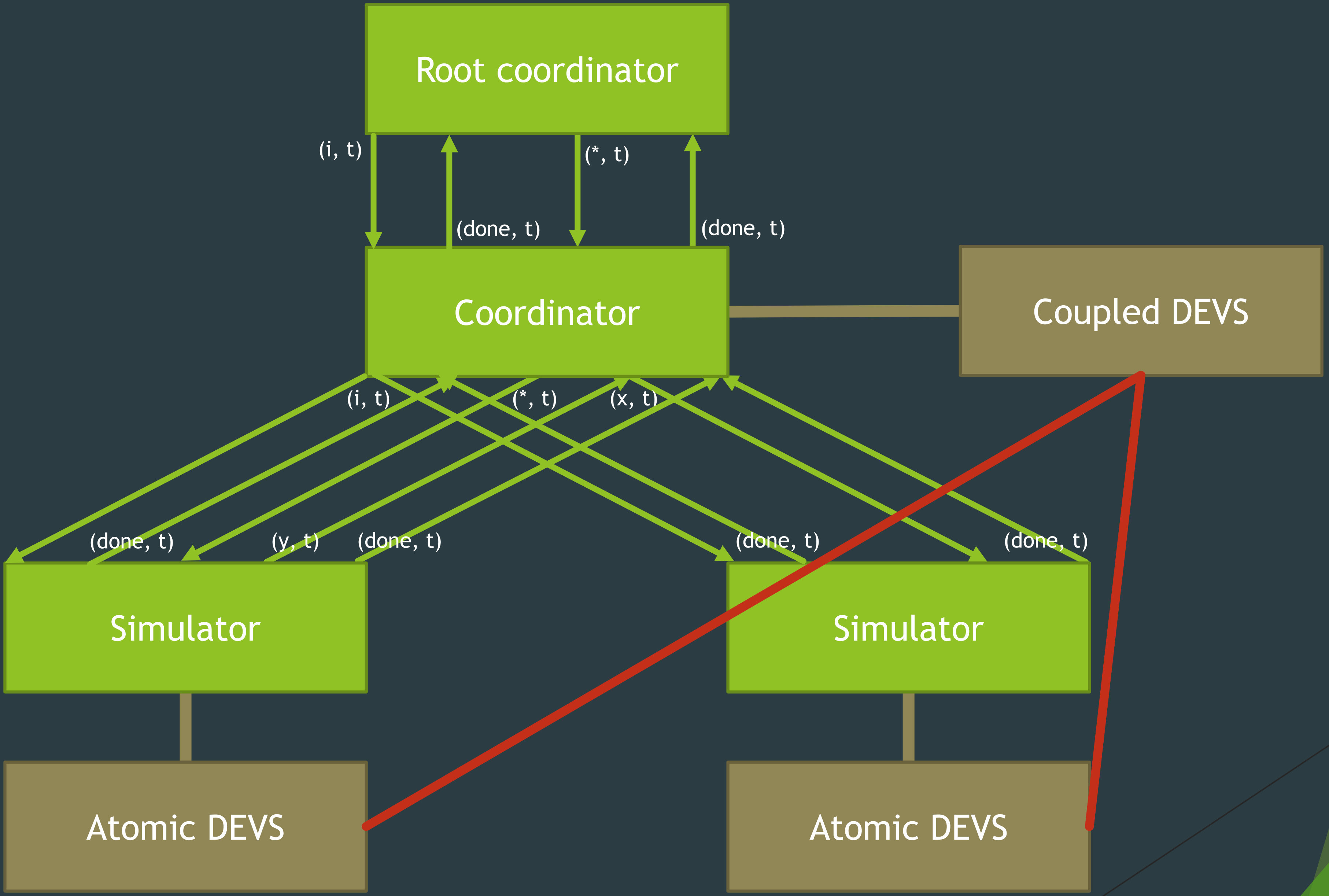






$$CM = \langle X_{self}, Y_{self}, D, MS, IS, ZS \rangle$$

$$flatten(CM) = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$


Root coordinator
(i, t)
(done, t)
(*, t)
(done, t)
Coordinator
Coupled DEVS
(i, t)
(*, t)
(x, t)
(done, t)
(y, t)
(done, t)
(done, t)
(done, t)
Simulator
Simulator
Atomic DEVS
Atomic DEVS

DEVS Semantics

	Operational Semantics	Denotational Semantics
Atomic DEVS	Abstract Simulator	[1]
Coupled DEVS	Hierarchical Simulator	Closure under Coupling

[1] Ashvin Radiya and Robert G. Sargent. A logic-based foundation of discrete event modeling and simulation. ACM Transactions on Modeling and Computer Simulation, 1(1):3-51, 1994.