Modelling of Software Intensive Systems

Assignment 6: DEVS

January 03, 2024

Thomas Gueutal
s0195095@ad.ua.ac.be

Felix Vernieuwe
s0196084@ad.ua.ac.be

Outline

0. Introduction

This report details our findings and solutions for assignment 6, concerning the implementation of a set of requirements into the Discrete-Event System Specification formalism. The assignment is made for the course *Modeling of Software-Intensive Systems* at the University of Antwerp.

0.1. PythonPDEVS

The **PythonPDEVS** python package is used for the specification of DEVS models.

0.1.1. Constraints

In order to follow the correct DEVS formalisms, the assignment kindly reminded us that if the Python-PDEVS documentation (or the lecture material) explicitly stated that something was not allowed (or oppositely, must be done), then we should by all means attempt to respect those constraints. To refresh the memory of the reader, but mostly to keep keep ourselves in check, we will try to maintain a list of relevant constraints in this section.

Next is a list of constraints that apply specifically to the `PythonPDEVS` package. So references to library-specific methods and variables will be used for the sake of brevity.

- **Atomic DEVS**
  - `extTransition(inputs)` : *"Should only write to the state attribute."*
  - `intTransition()` : *"Should only write to the state attribute."*
  - `outputFnc()` : *"Should **not** write to any attribute."*
  - `timeAdvance()` : *"Should ideally be deterministic, though this is not mandatory for simulation"*, and the assignment states that: *"The timeAdvance and outputFnc functions need to be deterministic and should **not** change the state of the model(s)."*
  - `__init__(self,)` : It is advisable to call `super().__init__(name,)` in the `__init__(self,)` method of every model class we write, where `name` is the unique name of the model within a (parent) coupled DEVS. If `name` is not provided to the AtomicDEVS or CoupledDEVS inherited class, then simulation may get confusing (go wrong?).

To guarantee good simulation results and readable code, we should respect some "soft" constraints.

- **SI units**: As per the assignment: "*Notice the **inconsistencies** used in units (minutes, seconds, kilometers, meters…). It is up to you to do a conversion to SI Units. This way, you can be ensured that no undesired side effects occur from invalid conversions.*"
- **Randomness**: As per the assignment: "*It is preferable to use a **predefined seed** for random number generation. This allows a consistent result when testing and debugging your code. Note that random number generators also do change a state and should therefore only be used where state changes are allowed!*
  PythonPDEVS comes with an RNG, yet, for the purposes of the assignment Python's or numpy's random will suffice."
- **Component Implementation**: As per the assignment: "*While all components of the model are defined above, **you are free to choose how these are implemented. As long as the behaviour is equivalent** to how it is described here. Sometimes, there are multiple solutions to the same problem, but there are also ambiguities to allow you to make choices in the implementation. Don't choose for the most impressive solution, but rather the easiest and fastest to implement. Don't forget to **discuss where and why** you made certain choices!*"
- **Model Versions**: "*Where possible, use **different experiment files to setup your tasks**. This allows you to return to previous tasks without issues, as well as referring to specific modules in your report. Also ensure your model is flexible enough to allow all the tasks by simply changing model parameters. If a task requires a (slight) change in the components used, make sure to include all versions of the components.*"
- **Classic DEVS**: As per the assignment: "*Your model should work for Classic DEVS, therefore you can **ignore the possibilities of parallelism**.*"

0.2. A Note on Polling Behavior

The assignment mentions the need for polling behavior for two components:

1. RoadSegment: when the velocity of the contained Car is 0.
   - `RoadSegment.Q_send` : Sends a Query as soon as a new Car arrives on this RoadSegment (if there was no crash). Additionally, a Query is sent every observ_delay time if the Car's v equals 0.
2. GasStation: When a QueryAck with `t_until_dep == 0.0` arrives.
   - `GasStation.observ_delay` : The interval at which the GasStation must poll if the received QueryAck has an infinite delay. Defaults to 0.1

Polling behavior is described similarly in both cases. A Query needs to be sent every `observ_delay` while a condition is met. We consider two possible implementations for this, based on the following scenario:

- Send 3 total Queries.
- Use a polling delay of 1s. This is interpreted differently for each polling implementation.
- Use a `observ_delay` of 10s. This is the lag between sending a Query and receiving the corresponding QueryAck.

We will refer to the first option as **realistic polling**. Multiple Queries and their QueryAcks may be in circulation at the same time. So the components sending Queries must also be capable of receiving and processing multiple QueryAcks in quick succession.

```
global time       action        event

0s                send          Query(ID=0)
1s                send          Query(ID=1)
2s                send          Query(ID=2)
10s               receive       QueryAck(ID=0)
11s               receive       QueryAck(ID=1)
12s               receive       QueryAck(ID=2)
```

The second option is **simlified polling**. Only one Query will be in circulation at a time. Once the Query is sent, the component becomes idle until the QueryAck is received. If the conditions is not fulfilled yet, the next Query is sent after waiting the polling delay.

```
global time       action        event

0s                send          Query(ID=0)
10s               receive       QueryAck(ID=0)
11s               send          Query(ID=1)
21s               receive       QueryAck(ID=1)
22s               send          Query(ID=2)
32s               receive       QueryAck(ID=2)
```

We believe this behavior most close

1. Model Design (I)

1.1. Messages

The messages exchanged are `Car` , `Query` and `QueryAck` . They represent events that are passed around through the system.

Each message was to be implemented as a **Python DataClass**, with its implementation located in the `components/messages.py` file. Note that **only some of the fields** of each dataclass were assigned default values. This is to ensure that at least a minimum of fields are meaningfully filled in by the caller of the constructor.

1.1.1. Car

A Car represents cars that drive through the road system. They are implemented as events that are passed from input to output port between different DEVS models. Cars will enter the simulation by being generated by Generator Atomic DEVS models. They leave the simulation when they enter a Collector Atomic DEVS component *or* when they crash in a RoadSegment.

This is implemented as the Car dataclass. This class has four constructor parameters that are not assigned any default values.

```
@dataclass
class Car:
    # Non-initialized members
    ID: UUID | int
    v_pref: float
    dv_pos_max: float
    dv_neg_max: float

    # Initialized members
    departure_time: float = 0.0
    distance_traveled: float = 0.0
    v: float = None     # Default value None, so we can default to v_pref
    no_gas: bool = False
    destination: str = ""
    source: str = ""
```

Codeblock 1: Car definition

This is done to ensure these parameters are assigned sane values.

1. Appropriate `ID` values promote readable verbose simulation output.
2. Variably determined `v_pref` values ensure dynamic interactions between Cars. This also reflects the non-uniformity of different preferred driving speeds in real-world settings.
3. Properly chosen `dv_pos_max` and `dv_neg_max` values reflect the different capabilities of vehicles in real-world settings.

The assignment noted the following for `dv_neg_max` and `dv_pos_max`: "*The maximal amount of respectively deceleration and acceleration possible for this Car on a single RoadSegment. Notice that this is not actually a velocity, but more of a velocity delta.*" We came up with two different interpretations.

1. since the terms acceleration and deceleration are used, '*velocity delta*' refers to SI unit $m/_{s^2}$, and we need to multiply this value by some Δt when a Car's velocity should change.
2. '*velocity delta*' simply refers the SI unit $\frac{m}{s}$, because the RoadSegment component description does not mention any such time delta.

Our implementation **assumes the second interpretation**. This results in 'immediate' velocity changes.

The by-default members are initialsied such that their true default values are either assigned later, or they start off with neutral values that can be updated throughout the simulation. We pay special attention to:

1. `v` its default is `None`. Though this may seem strange at first glance, it allows the dataclass' `__post_init__()` method to recognize that no initial speed was been provided, such that

the velocity can be initialized to `v_pref`. As per the assignment: "*v (float): The current velocity. By default, it is initialized to be the same as v_pref, but may change during the simulation.*"
2. `destination` defaulting to an empty string is actually unsafe, if the Car were to have to interact with a CrossRoads type of DEVS model, since its destination would never be reached. However, for compatibility reasons with the provided basic test suite, it had to be assigned *some* default value.
3. `source` represents the source DEVS model that the Car originated from. For this assignment, this should always be a Generator. Its main use is for debugging. Further, it is initialized to an empty string to ensure compatibility with the tests.

1.1.2. Query

A Query represents a Car (driver) looking from one RoadSegment into the next one, checking if it is already occupied. After sending a Query, at some point in the future a QueryAck should be sent in reply to the Query. This may be intuitevly understood as a ray of light bouncing off a Car in the next segment and ending up in the eye of the driver of the Car. To mimic real-world reaction times and weather conditions, the components that should reply to any received Queries can customize the delay with which they reply to said Queries.

```python
@dataclass
class Query:
    # Non-initialized members
    ID: UUID | int

    # Initialized members
    source: str = ""
```

Codeblock 2: Query definition

Note that the Query dataclass also makes use of a `source` parameter. This is again for debug purposes. It indicates which DEVS model broadcasted the Query to all models connected to its 'Query send' port. This too is initialised to an empty string in order to guarantee compatibility with the testing suite.

1.1.3. QueryAck

A QueryAck should always be sent in response to a Query arriving. They serve to inform the querying Car of when the target RoadSegment will become unoccupied, or if it already is empty. As per the assignment: "*QueryAck – Event that answers a Query. This is needed to actually obtain the information of the upcoming RoadSegment. The Query/QueryAck logic can therefore be seen as "polling".*"

To mimic real-world reaction times and weather conditions, the components that should reply to any received Queries can customize the delay with which they reply to said Queries.

```python
@dataclass
class QueryAck:
    # Non-initialized members
    ID: UUID | int
    t_until_dep: float

    # Initialized members
    lane: int = 0
    sideways: bool = False
    source: str = ""
```

Codeblock 3: QueryAck definition

The parameters of note are:

1. `t_until_dep` is an estimate of how long it will take for the Queried RoadSegment to become free, **in seconds**.
2. `lane` refers to which road lane the QueryAck is intended. This relates to the principle of lane switching within our system. The Fork component implements this behavior. By inspecting the lane of incoming QueryAcks, the Fork determines which QueryAcks to ignore.
3. `source` represents the source DEVS model that broadcast the QueryAck from its QueryAck send port. Its main use is for debugging. We also initialise this to zero for the test suite.

1.2. Road Segment

1.2.1. General Description

A RoadSegment is a stretch of road that can contain **at most one Car**. Conforming to the reality set by the assignment, if a second Car enters the segment at any point while a Car is already present, both Cars are immediately vaporized into their constituent atoms; a crash will remove all of the involved Cars from the traffic simulation.

Following this assumption, we make the claim that considerations related to the RoadSegment containing more than one Car in its `RoadSegment.state.cars_present` state member, can be safely ignored for our implementation.

All the logic for entering a Car into a RoadSegment is encapsulated in the `RoadSegment.car_enter(Car)` method. This method may be used to **pre-fill a RoadSegment** with a Car. The logic to exit the current Car from the RoadSegment and reset the segment's state, is encapsulated in the `RoadSegment._exit_car()` method. Do note that this method requires a Car to be present, else an exception will be raised.

1.2.2. Query Receival

Roadsegments are the principal receivers of Queries. This is evidently because most components of traffic infrastructure is roads. Each segment has a personal `RoadSegment.state.incoming_queries_queue` member. This is a FIFO queue of incoming Queries from other components, that need to be responded to using a QueryAck. Because every Query is enqueued for the exact same time, a FIFO queue suffices. Each Query is stored along with a stopwatch timer, to keep track of when to broadcast the related response.

Every QueryAck that is sent, should contain the up-to-date `t_until_dep` value of the RoadSegment. Since the `outputFnc` method may **not** edit the state, this value needs to be dynamically calculated based on the `timeAdvance()` value. This is done by chaining the helper methods `RoadSegment._calc_updated_remaining_x(float)` and `RoadSegment._calc_updated_t_until_dep(float)`.

The Query receival + QueryAck response behavior does not impact the handling of the Car within the RoadSegment.

1.2.3. Timers

All timers used within the component are updated at the start of the `intTransition()` and `extTransition(inputs)` methods, using the `RoadSegment._update_multiple_timers(float)` method, using `timeAdvance()` and `self.elapsed` respectively. For each timer that is decremented in this update step, we made sure to round the decreased timer value up to 0.0 using a max function call. This is because floating point errors related to decrementing by `self.elapsed` can make the timers appriximately, but not exactly 0.0, which can mess up some of our boolean checks that rely on the timers reaching 0.0.

1.3. Generator

The Generator component/model leaves some ambiguity, so we have some design decisions to make.

First of all, we determine the deterministic set of actions the generator makes. For this we refer back to the descriptions of the Generator model in the assignment.

*General Component Description – ... When a Car is **generated**, a Query is **sent** over the Q_send port. As soon as a QueryAck is **received**, the generated car is **output** over the car_out port. Next, the Generator **waits** for some time **before generating** another Car. ...*

We highlighted in bold the words that lead to actions within the generator. This general description provides a clear, deterministic (sequential) series of actions for the generator to perform. **But**, the description of the Q_rack input port contradicts the "*... As soon as a QueryAck is received...*" in the general description:

*Q_rack – Port that **receives** QueryAck events. When such an event is received, the next Car will be **outputted after the QueryAck's t_until_dep time** has passed. This way, there will not be any crashes due to a Generator.*

So we must amend the steps that the general description provides, with a waiting step after the QueryAck is received, because "*the next Car will be outputted after the QueryAck's t_until_dep time has passed.*"

*car_out – **Outputs** the newly generated Car. **The current simulation time becomes the Car's departure_time**.*

We must be **very** careful when implementing this step. The *outputFnc()* method is *not* allowed to alter the state of the DEVS model. The current simulation time will have to be assigned during a previous step, while taking into account the *timeAdvance()* that is expected to occur before this output step.

*Q_send – **Sends** a Query as soon as the newly sampled **IAT says so**.*

This description essentially reveals the circular nature of the deterministic series of steps of the generator.

What follows is the sequential list of generator steps derived from the above descriptions. Each step is annotated with the DEVS method they are associated with. This concretizes each step.

1. generate a Car	*intTransition*
2. send a Query on Q_send port	*outputFnc*
3. **wait** for INFINITY seconds / be idle	*timeAdvance*
4. receive a QueryAck on Q_rack port	*extTransition*
5. **wait** for QueryAck.t_until_dep seconds	*timeAdvance*
6. send the Car on car_out port	*outputFnc*
7. **wait** for IAT seconds	*timeAdvance*
8. goto step 1.	

As a conclusion, we discuss our interpretation of the **limit parameter** of the Generator DEVS model. This matters, because the Generator may be implemented as generating new Cars while the previous ones await leaving the generator **or** as only generating a new Car after the current one has left. The first interpretation requires that we store a queue of Cars and that we ensure the IAT timer is not invalidated by external events (pattern 1: ignore event). The second interpretation does not require those considerations, as all steps are purely deterministic and sequential.

Seeing as we provided a sequential list of steps, we naturally **will assume the second interpretation**. It is impossible for the external transition, receiving a QueryAck, to interrupt another timer, because the Generator remains idle while waiting for the QueryAck. So the timeAdvance is INFINITY.

The limit parameter is described as follows: "Upper limit of the number of Cars to generate." Due to our sequential steps interpretation, we assume this to mean that **after generating limit Cars, the Generator will become idle forever** and stop producing any Cars from the on.

1.4. Collector

1.5. SideMarker

1.6. Fork

1.7. GasStation

The GasStation component is described as follows.

General Component Description – Represents the notion that some Cars need gas. It can store an infinite amount of Cars, who stay for a certain delay inside an ***internal queue****.*

car_in – Cars can enter the GasStation via this port. As soon as one is entered, it is given a delay time, sampled from a normal distribution with mean 10 minutes and standard deviation of 130 seconds.

Cars are required to stay at least 2 minutes in the GasStation. When this delay has passed ***and when this component is available****, a Query is sent over the Q_send port.*

Q_rack – When a QueryAck is received, the GasStation waits for QueryAck's t_until_dep time before outputting the next Car over the car_out output. ***Only then, this component becomes available again****. If the waiting time is infinite, the GasStation keeps polling until it becomes finite again.*

Q_send – Sends a Query as soon as a Car has waited its delay. ***Next, this component becomes unavailable****, preventing collisions on the RoadSegment after.*

car_out – Outputs the Cars, with no_gas set back to false.

We give special attention to the word **queue** in the general description. The specification details in the assignment (i.e. the descriptions above) introduce a notion of "availability of the component". This is to prevent collisions due to a Car leaving the GasStation through the car_out port onto the connected RoadSegment.

When a single Query is sent on the Q_send port, the GasStation component becomes unavailable. This may only be done if the component was available before sending the Query. Only when a Car is output on the car_out port, does the component become available again. Note that outputting a Car onto the connected RoadSegment means that RoadSegment is now occupied. So for the next Car, a new Query must be sent out again to prevent collisions.

This indicates that the **GasStation does output Cars one by one, using a queue**.

We will make use of a **priority queue**, instead of for example a FIFO queue. The key for the priority queue will be the remaining (refuel) delay time of each Car in the queue, where a lower remaining delay time means higher priority. So, Cars that entered the GasStation sooner but got assigned a long delay time may leave after Cars that entered later but were assigned a short delay time. Furthermore, if some Car with ID x arrived earlier than some Car with ID y, and after some time both are still in the queue and reach a refuel delay time of 0.0s, then Car y may still depart before Car x is allowed to depart, as a result of our choice of key for the priority queue. In short, **we do not guarantee that the arrival order of the Cars in the GasStation has any influence on their departure order**.

The choices discussed above were made to allow an easier, less complicated implementation.

1.8. CrossRoad (Segment)

2. Coupled DEVS

2.1. RoadStretch implementation

2.2. RoadStretch simulation (II)

In this section, we will simulate a Coupled DEVS model, that connects a generator to a collector, using a series of RoadSegment pieces chained together, with a split in the middle of the road — containing a gas station.

We implemented this model by first creating each of the RoadSegments, and then stitching them together according to a given layout. This allows for easy extensibility of the road layout, if so desired. The amount of road segments between the Generator and Fork, and Merge and Collector, can be varied by changing the seg_{gen} and seg_{col} variables (within the model given as `generator-` and `collector_segment_count` respectively).

For the following sections, we will run the simulations using following parameters, these will — unless explicitly stated otherwise — remain unchanged. It is also important to state that the assignment-set values are *not* included (such as `dv_pos_max`).

Parameters:

Individual road length	Generator limit	Observe delay
$L = 10.0m$	$\tilde{l} = 10$	$Q_{\text{delay}} = 0.1s$

Max. velocity main	Preferred velocity avg.	Inter-Arrival-Time$_{\text{min}}$	Inter-Arrival-Time$_{\text{max}}$
$v_{\text{main}_{\text{max}}} = 12.0\,{}^{\text{m}}\!/_{\text{s}}$	$v_\mu = 15.0\,{}^{\text{m}}\!/_{\text{s}}$	$\text{IAT}_{\text{min}} = 10.0s$	$\text{IAT}_{\text{min}} = 18.5s$
Max. velocity offroad	Preferred velocity S.D	#Generator segments	#Collector segments
$v_{\text{offroad}_{\text{max}}} = 5.0\,{}^{\text{m}}\!/_{\text{s}}$	$v_\sigma = 1.0\,{}^{\text{m}}\!/_{\text{s}}$	$\text{seg}_{\text{gen}} = 2$	$\text{seg}_{\text{gen}} = 2$

In order to get rid of any run-to-run variance between bespoke simulations, we chose to simulate the environment several times, averaging out general statistics. In the configurations, this will be given as n.

2.2.1. Short simulation (A)

For the short simulation, we chose following run parameters:

Parameters:

Simulation length	Generator limit	Runs generated
$t = 20.000s$	$\tilde{l} = 200$	$n = 30$

Our main benchmark for the simulation will be based upon the crash rate, defined as:

$$\text{arrival rate} = \frac{\text{amount of arrivals}}{\text{amount of departures}} \qquad \text{crash rate} = \frac{\text{amount of crashes}}{\text{amount of departures}}$$

The “amount of arrivals” represents the amount of cars that arrived at the Collector in the end of the RoadStretch (`collector.state.n`), and “amount of departures” similarly represents the amount of cars having been generated in the

2.2.2. Long simulation (B)

Parameters:

Simulation length	Generator limit
$t = 5.000.000s$	$\tilde{l} = 4.000$

2.2.3. Observations (B-C)

2.3. 4-Way CrossRoads implementation

2.4. 4-Way CrossRoads simulation (III)

2.4.1. Order of execution (A)

2.4.2. Right-of-Way intersection (B)

2.4.3. Roundabout intersection (C)