Modelling of Software Intensive Systems

Assignment 6: DEVS

January 04, 2024

Thomas Gueutal
s0195095@ad.ua.ac.be

Felix Vernieuwe
s0196084@ad.ua.ac.be

Outline

0. Introduction **2**
0.1. Important Notes 2
0.2. Changes since last submission 2
0.3. PythonPDEVS 3
0.3.1. Constraints 3
0.4. A Note on Polling Behavior 4
0.4.1. Realistic Polling 4
0.4.2. Simplified Polling 4
0.4.3. Conclusion 5
0.4.4. Extra: Realistic Polling Implementation 5
0.5. A Note on Timers and TimeAdvance 6
0.5.1. Problem: Negative TimeAdvance 6
0.5.2. Solution: Use Max Function 7

1. Model Design (I) **8**
1.1. Messages 8
1.1.1. Car 8
1.1.2. Query 9
1.1.3. QueryAck 9
1.2. Road Segment 10
1.2.1. General Description 10
1.2.2. Query Receival 10
1.2.3. Timers 11
1.2.4. Velocity Updates 11
1.2.5. Polling 11
1.3. Generator 12
1.4. Collector 13
1.5. SideMarker 13
1.6. Fork 13
1.7. GasStation 14
1.8. CrossRoadSegment 15
1.9. CrossRoads 15

2. Coupled DEVS (II-III) **16**
2.1. RoadStretch Simulation (II) 16
2.1.1. Short Simulation (A) 17
2.1.2. Long Simulation (B) 18
2.1.3. Observations (B-C) 19
2.1.4. Order of Execution 19
2.2. 4-Way CrossRoads Simulation (III) 21
2.2.1. Regular Intersection (A) 21
2.2.2. Right-of-Way Intersection (B) 22
2.2.3. Roundabout Intersection (C) 24
2.2.4. Intersection Type Comparisons (D) 25

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, and then achieving a better understanding by simulating and analysing two different coupled components.

The assignment is made for the course *Modeling of Software-Intensive Systems* at the University of Antwerp, with Randy Paredis as the contact person for the assignment.

0.1. Important Notes

For this assignment, no additional packages need to be installed, aside from the not-included PyDEVS package.

All drawings and diagrams are generally also included in the report, in the caption for each image, you will be able to find the TAPAAL file where the simulatable diagram can be found (this is given as TAPAALFILE.TAPN/COMPONENT). In general, you will be able to find the TAPAAL files under `~/models/*`.

Further, each diagrams is also rendered as a SVG directly into the PDF, so they should be infinitely zoomable. In case this does not work properly, the diagrams are also separately available in the `~/report/diagrams/*` folder of the submission archive, following the same naming scheme as the DEVS components they're based on.

When we make hypotheses and/or assumptions in a task, we clearly mark these by prefixing **ASSUMPTION**/**HYPOTHESIS** to it.

For navigating the project and figuring out to which task each (sub)section belongs to, we have suffixed either roman numerals (I) and alphabetic characters (A) to the relevant sections. These correspond to the numbers (1.) and sub-bulletpoints (◦) found in the `Tasks` section of the assignment.

0.2. Changes since last submission

POSTSCRIPTUM NOTE: As always, we severely underestimated the amount of time it takes to write up the report. *Almost* no changes were made to the code itself, except for:

- Fix Right-of-Way CROSSROADS ports connections being incorrectly bound
- Adapted simulation parameters to be ever so slightly more realistic
- Added code for generating graphs (plus additional statistics)

The most drastic change for the new submission is the report, including:

- Missing sections added (incl. simulation conclusions)
- Fixed spelling mistakes

0.3. PythonPDEVS

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

0.3.1. Constraints

In order to follow the correct DEVS formalisms, the assignment kindly reminded us that if the PythonPDEVS 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**.*"

0.4. 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#sub[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.

0.4.1. Realistic Polling

We will refer to the first option as **realistic polling**. See Figure 1 for a simple sequence diagram. Every 1s of global time a Query is sent. After the third Query is sent, all QueryAcks are still enroute. In the actual implementation, you would only stop sending Queries after the polling termination condition is reached. But for the sake of this example, we stop after only three Queries.

The components sending these Queries must also be capable of receiving and processing multiple QueryAcks in quick succession or even within the same exact moment of global time. After the polling termination condition is reached, it may be necessary to ignore all following QueryAcks, even if some of those acks would have been accepted were polling still active.

```
global time       action        event               termination reached?

0s                send          Query(ID=0)         False
1s                send          Query(ID=1)         False
2s                send          Query(ID=2)         False
10s               receive       QueryAck(ID=0)      False
11s               receive       QueryAck(ID=1)      True
12s               receive       QueryAck(ID=2)      True ==> ignore QueryAck?
```

0.4.2. Simplified Polling

The second option is **simplified polling**. See Figure 2 for a simple sequence diagram. 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 termination condition is not fulfilled yet, the next Query is sent after waiting the polling delay.

The components sending these Queries must still be capable of receiving and processing multiple QueryAcks in quick succession or even within the same exact moment of global time. This is because Queries are often broadcast to multiple receiving components in more complex coupled DEVS models. So, after the polling termination condition is reached, it may again be necessary to ignore all following QueryAcks, even if some of those acks would have been accepted were polling still active.

```
global time       action        event               termination reached?

0s                send          Query(ID=0)         False
10s               receive       QueryAck(ID=0)      False
11s               send          Query(ID=1)         False
21s               receive       QueryAck(ID=1)      True ==> no Query(ID=2)
```

0.4.3. Conclusion

We believe realistic polling most closely follows the polling behavior described in the assignment. However, **we ended up going with simplified polling** for two major reasons.

1. The provided test cases do not seem to support realistic polling. The provided test cases make use of the PingPong helper Atomic DEVS model. This model only has support for one Query at a time. Making use of realistic polling causes the test to run infinitely, which may mean that the full test suite used for grading our assignment solution could fail/run infinitely as well.
2. Solving the assignment easier. becomes easier. After implementing realistic polling for GasStation, we felt that the and added complexity and implementation time of adding it was not worth it, see Section 0.4.4.

We expand more on the **first argument**. The PingPong model only has room for one QueryAck at any time. Given the default polling delay (0.1s) and `observ_delay` (0.1s) specified in the assignment, the PingPong model often has its internally stored Query overwritten before a QueryAck can be generated. This causes the `PingPong.timeAdvance()` to always be reset to 0.1. This happens because input events to the PingPong model are processed by `PingPong.extTransition()` before the internal transition that the `PingPong` model would otherwise have generated using `PingPong.outputFnc`.

0.4.4. Extra: Realistic Polling Implementation

0.4.4.1. What

We did implement a version of realistic polling for the GasStation component before realising this problem. For this, see the file `other/gasstation_realistic_polling.py`. It is worth mentioning that this version of the GasStation may differ significantly from our 'official' implementation of it in `components/gasstation.py`; we did not maintain the realistic polling version of GasStation after we reverted to using simplified polling.

0.4.4.2. How

When we implemented realistic polling, we made an attempt to **generalize the polling behavior into a reusable interface**. This interface, the QueryPollingState class in `other/query_polling_state.py`, should encapsulate the polling state and methods needed to provide polling for GasStation, RoadSegment and optionally Generator if we felt the need. Importantly, the no logic related the start or termination conditions of polling are provided by it. This had to be handled outside this interface, because GasStation and RoadSegment different start and termination conditions for when (not to) poll.

The **GasStationState** of the realistic polling implementation **inherits from this interface**. The interface methods are exclusively utilized to perform polling; the polling state should not be manually accessed.

0.4.4.3. Experiment

To support the testing, we copied and extended the PingPong model into a PingPongMulti model. It is essentiall a PingPong model with support for receiveing, storing and responding to any amount of Queries. See `other/ping_pong_multi.py`. The PingPongMulti constructor lets you define a few parameters:

- `t_until_dep`: The `QueryAck.t_until_dep` for all generated QueryAcks
- `pong_delay`: The observ delay between Query receival and QueryAck sending
- `first_x_inf`: The `first_x_inf` number of generated QueryAcks are assigned a `QueryAck.t_until_dep` equal to $+\infty$ instead of the specified `t_until_dep`
- `do_max`: The whether to simply decrement all internally kept timers in PingPongMulti, or to clampt their lower bound to 0.0

```
class PingPongMulti(AtomicDEVS):
      def __init__(self, name, t_until_dep: float = 0.0, pong_delay: float = 0.2,
                   first_x_inf: int = 0, do_max: bool = False):
            super(PingPongMulti, self).__init__(name)
      ...
```

To test the veracity of our realistic polling, an experiment was created. See `experiments/negative_time_advance.py`. There, a PingPongMulti is hooked up to a RoadSegment. The no true assertions are used, the verbose trace output has to be manually inspected.

0.5. A Note on Timers and TimeAdvance

We decided to model our use of multiple timers based on the multiple timers pattern described in the course slides. This involves:

1. Using INFINITY as a neutral value for unused timers (timers that are OFF)
2. Always decreasing/increasing timers by
    - `elapsed` at the start of `extTransition`, to mitigate the interruption of an external event (pattern 1: ignore an event)
    - `timeAdvance()` at the start of `self.intTransition()` because an internal event occurred

So, initialising and resetting timers often makes use of $+\infty$.

To produce the final `timeAdvance()` output, we take the minimum of all timers that should lead to some internal event.

0.5.1. Problem: Negative TimeAdvance

We did **encounter a problem** with this implementation. Sometimes, the `timeAdvance()` output would be negative. Namely, the pypDEVS library throws a DEVS exception when the timeAdvance value is negative. We had to take this into accound due to how we implement our DEVS models.

But how or when is this possible? Our unused timers are always set to $+\infty$, so decreasing those by any amount should not result in a negative value. Thus only timers that may cause issues are 'active/running' timers. But all active timers lead to some internal event, so they are all considered in the time advance function to select the lowest timer value of them all. So every time the `intTransition()` method is called and the timers are decreased by it, they are decreased by the lowest, exact timer value, which should not lead to a negative value. In the 'worst case' multiple timers become 0.0.

The only cause we can think of, and which we verified sometimes happened, was that the `self.elapsed` value available in the `extTransition()` method was a tiny amount larger than the lowest timer, resulting in a timer that was decreased to below 0.0, which in turn caused a negative time advance later.

So we presume this problem was caused by floating point rounding errors. To illustrate this problem, see `experiments/negative_time_advance.py`. The MultiPingPong class has support for enabling or disabling the use of a max function to clamp the Query observ delay timers to 0.0 as a lower bound. **However, the `DO_MAX_FOR_EXT` variable in that experiment should manually be set to False before running, else no negative time advance will ever occur!** This is because this experiment also showcases our realistic timer implementation, which should run without encountering negative time advance errors.

As a finaly note, the `supermarket.py` file written during the demo also shows this problem. If you run that file a handful of times, a negative time advance DEVS exception will eventually be thrown.

0.5.2. Solution: Use Max Function

We make use of a simple if hacky solution: always **clamp timers to above 0.0** if they are decreased during timer updates.

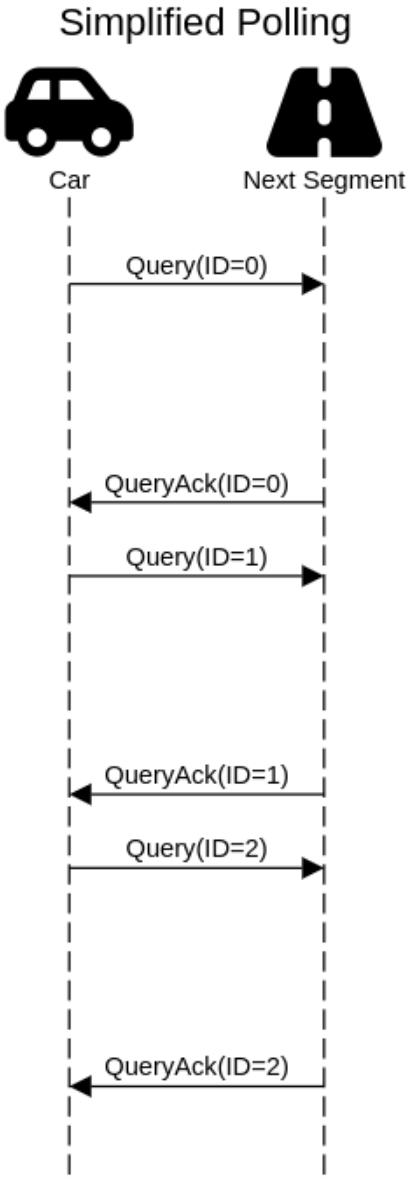

Figure 1: It takes 10s observ_delay to receive QueryAck, Wait 1s polling delay after receiving QueryAck before sending next Query

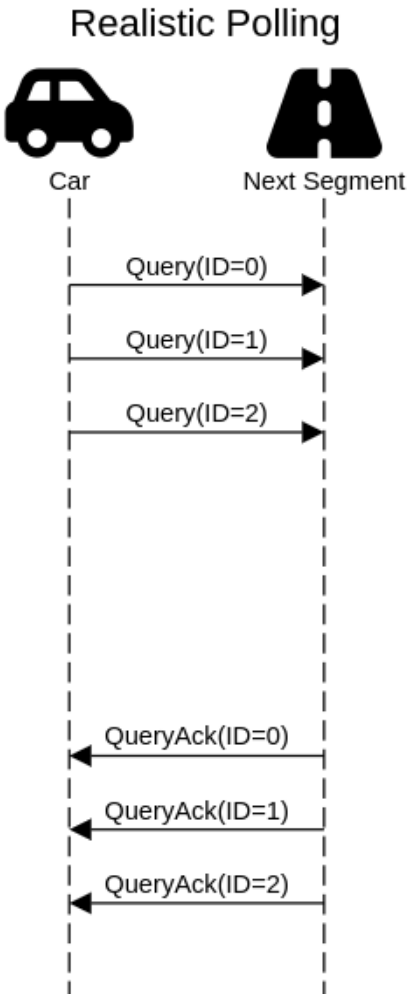

Figure 2: It takes 10s observ_delay to receive QueryAck, Wait 1s polling delay after sending Query before sending next Query

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.

```python
@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, by `timeAdvance()` and `self.elapsed` respectively. For each timer that is decreased in this update step, we made sure to round the decreased timer value up to 0.0 using a max function call. See **Section 0.5.** on timer design and **Section 0.5.2.** on the use of max.

The `RoadSegment.state.t_until_dep` value represents an estimation of how long it will take for the current Car to leave the RoadSegment, in seconds. If it is 0.0, then the RoadSegment is unoccupied. We update this value as if it were a timer in the DEVS model. This is appropriate, because when we update it, we also recalculate `RoadSegment.state.remaining_x`. This allows the velocity updates due to arriving QueryAcks to correctly determine a new Car velocity.

1.2.4. Velocity Updates

Updating the velocity of the Car in a RoadSegment only happens in response to a QueryAck. Such a QueryAck must in turn have been sent in response to a Query that the RoadSegment in question emitted earlier.

To correctly update the velocity at a single global time, multiple QueryAcks may need to be considered. For example, at global time 14.514, three QueryAcks, with IDs 1, 1 and 2, may be waiting to be applied to the RoadSegment containing the Car with ID 1. The ack with ID 2 is ignored due to ID mismatch.

The other two acks are both processed. Each of them results in a distinct updated velocity value. However, the `QueryAck.sideways` boolean field and the `RoadSegment.state.priority` boolean member combined describe the traffic-related circumstances of each velocity update value. This gives rise to an ordering of the velocity updates, irrespective of the actual new velocities themselves.

This ordering is reflected in the Priority enum, located in `components/roadsegment.py`.

Lastly, we compute the final velocity update in the same global time step by keeping track of the current updated velocity's priority designation. Only if a new velocity value has a priority greator or equal to the current one, can the velocity conditionally overwritten. e.g. Before the two QueryAcks have been applied, the Car's velocity is $10\frac{m}{s}$. QueryAck one results in a new velocity of $0\frac{m}{s}$, with a priority value of P2. The second QueryAck would result in an updated velocity of $15\frac{m}{s}$, but it has priority P0. So the new velocity is $0\frac{m}{s}$. After the current moment concludes, i.e. any amount of time passes, the `RoadSegment.state.v_current_priority_int` is set to priority PNone, meaning it is unset.

1.2.5. Polling

When a Car enters a RoadSegment, a Query is sent to optionally update the Car's velocity. Note that this Query is sent *immediately* upon the arrival of the Car, if no collision occurred. If the received QueryAck causes the Car's velocity to be changed to 0.0, then polling should start.

For our polling implementation, refer back to **Section 0.4.2.**. Since we make use of simplified polling, A Car will actually receive a QueryAck every `polling_delay + observ_delay` seconds.

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.

In the unlikely scenario that our v_pref sampled from the normal distribution is lower or equal to 0.0 (due to configuration or RNG), we fall-back our v_pref to a known value to prevent having deadlocks (for 0.0 speed), or errors (for negative speed). See the v_pref_fallback parameter of the `Generator.__init__(...)` constructor

1.4. Collector

The description for the Collector component is briefly given as:

> *General Component Description – Collects Cars from the simulation and stores all important information such that statistics can be computed afterwards.*

In essence, every time a Car enters, we need to update some statistics. This is nothing more than executing a state update when receiving a `car_in` for `extTransition(...)`. In terms of statistics, aside from keeping the mandated statistics (`n`), we also keep additional statistics for the simulations that follow in later sections: `n_no_gas` , `latest_arrival_time` , `car_travel_stats` (time in travel, distance travelled), `total_refuel_time` , `n_times_refuelled` .

1.5. SideMarker

The SideMarker in itself is a very simple component, defined as:

> *General Component Description – Marks all inputted QueryAcks to have sideways set to true. It also immediately outputs the inputted events.*

When the SideMarker receives a QueryAck on the mi port, it is set to `sideways` and enqueued for outputting in the immediate next timestep. As it is possible that multiple QueryAcks get sent to the SideMarker at the same time, we decided to store and pop them in/from a queue (as in RoadSegment). Every `intTransition` , the front QueryAck will be passed to every component connected to the mo port.

1.6. Fork

Next, the Fork will extend most of the logic from RoadSegment, with one additional property:

> *General Component Description – Allows Cars to choose between multiple RoadSegments. This allows for a simplified form of "lane switching".*

We simply defined the new output port car_out$_2$, and to prevent duplication of code, we first call the original RoadSegment code in order to determine whether a Car has reached the end of the road. If this is the case, then car_out will contain a Car that either has to go towards the top or bottom segment of the Fork.

Thus, after the original code, we just need to redirect the car_out output to car_out$_2$, if necessary (based on `no_gas`). Further, we also need to verify that the Car will only focus on QueryAck's from relevant segments, this can be easily done by checking whether the Segment `lane` is equal to `no_gas` (if `no_gas` , then you should only receive acknowledgements from the `no_gas` segments, given by lane 1).

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.

Note also that we will make the **assumption** that a Car maintains its speed when it enters the GasStation, as the requirements do not explicitly mention anything about the Car's velocity while it is tanking. (In a realistic environment, the Car should obviously be at a standstill while tanking, but then again, we also exist in a world where Cars can instantly atomize when they ever so slightly touch each other).

1.8. CrossRoadSegment

The CrossRoadSegment follows suit to the Fork component, slightly extending its original behaviour to allow for additional input/output ports for Cars:

> *General Component Description – Acts as if it were a part of a crossroads. I.e., this is a location where two roads merge and split at the same time.*

This component could actually be seen as a generalisation of the Fork: instead of routing based on the `no_gas` attribute, we now determine the output port via a more generic `destination` attribute. In this regard, then, car_out_cr is identical to car_out_2; using the exact same code to move a Car on the car_out port to car_out_cr if necessary (i.e. when the Car destination is not in the current CrossRoadSegment).

The one major difference between Fork and CrossRoadSegment, however, is that there is an additonal input port: car_in_cr. We may simply consider this to be equivalent to the car_in port of the regular RoadSegment. *Before* we run the `extTransition` of the RoadSegment, we will first do a `car_enter` to run the required logic for a Car arrival.

Note that we **assume** that two cars being on a CrossRoadSegment as the same time constitutes as a crash, just as it does for a RoadSegment — even if they technically arrive from different directions.

1.9. CrossRoads

The CrossRoads implementation was rather straight-forward, as the work consisted of connecting already defined components in a CoupledDEVS. The only note-worthy aspect of this component, is the fact that we needed to be careful about how we connected the output/input ports of the coupled component, with the ports of the individual CrossRoadSegments. Luckily, the diagram provided proved very useful for not accidentally connecting the wrong pair of ports.

2. Coupled DEVS (II-III)

2.1. 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, such as `dv_pos_max` and mean GasStation refuel time, also remain the same. See `other/constants.py` for several such values.

Parameters:

Generator limit		Observe delay
$\tilde{l} = 100$		$Q_{\text{delay}} = 0.1s$
Connecting segment length	Main segment length	Offroad segment length
$L_{\text{conn}} = 5.0m$	$L_{\text{main}} = 10.0m$	$L_{\text{offroad}} = 5.0m$

Max. velocity main	Preferred velocity mean	Inter-Arrival-Time$_{\text{min}}$	Inter-Arrival-Time$_{\text{max}}$
$v_{\text{main}_{\text{max}}} = 30.0\,\text{m}/\text{s}$	$v_{\mu} = 25.0\,\text{m}/\text{s}$	$\text{IAT}_{\text{min}} = 10.0s$	$\text{IAT}_{\text{min}} = 15.0s$
Max. velocity offroad	Preferred velocity S.D	#Generator segments	#Collector segments
$v_{\text{offroad}_{\text{max}}} = 20.0\,\text{m}/\text{s}$	$v_{\sigma} = 5.0\,\text{m}/\text{s}$	$\text{seg}_{\text{gen}} = 10$	$\text{seg}_{\text{coll}} = 10$

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.

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

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

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 generator. "amount of crashes" is calculated as the sum of the `RoadSegments.state.collisions` over all RoadSegments.

Further, we also collect statistics of the general travel time of each car — which both shows if we happen to have any traffic jams, and how smoothly cars can generally can travel.

2.1.1. Short Simulation (A)

For the short simulation, we chose following run parameters:

PARAMETERS:

Simulation length	GENERATOR limit	Runs generated
$t = 200.0s$	$\tilde{l} = 100$	$n = 30$

We interpreted *short simulation* to mean a simulation where not all CARS have the opportunity to leave the simulation. This way, there is little time for the road to become filled with CARS, so that more complex inter-CAR interactions lead to crashes.

In general, we can describe the minimum time spent in the system according to the following equation:

$$t_{\min} = \left(\frac{\left(\text{seg}_{\text{gen}} + \text{seg}_{\text{coll}}\right) \times L_{\text{main}}}{\min\left(v_{\text{main}_{\max}}, v_{\text{pref}}\right)} \right) + \begin{cases} \frac{3 \times L_{\text{main}}}{\min\left(v_{\text{main}_{\max}}, v_{\text{pref}}\right)} & \textbf{if has\_gas} \\ \frac{2 \times L_{\text{offroad}}}{\min\left(v_{\text{offroad}_{\max}}, v_{\text{pref}}\right)} + \max(150, \mathcal{N}(600, 150)) & \textbf{if no gas} \end{cases}$$

Note that this formula does indeed not account for:

1. CAR slowdown due to merges or traffic jams
2. Inter-Arrival-Time of generators being variable
3. Waiting for QUERYACK's

We can generally state, however, that there is a *greater risk* of collisions when a CAR remains relatively longer compared to other CARS on the ROADSEGMENTS that constitute the ROADSTRETCH. Because then there is more opportunity for faster CARS to close the distance, and potentially not have the time to slow down.

However, since the simulation is run only for a short time, there is no opportunity for congestion. This is reflected by the **large peak of speedy arrivals** in Figure 4. Around 40 of the CARS can reach the collector without much contention.

This is likely aided by the relatively **fast maximum allowed velocity ($30\frac{m}{s}$)** on each ROADSEGMENT, compared to the **fairly short length of the main ROADSEGMENTS ($5m$)** that connect the GENERATOR with the COLLECTOR.

Once the slower and faster cars start to be mixed in the together, they start to impede one another. This results in the tail of slightly longer travel times. Note that none of the arrivals here include cars that at some point switched lanes to the GASSTATION, as those need to wait a minimum of 120s in the GASSTATION.

All of this results in a collision rate of 0%. The fact that there are no cars trying to merge from the GASSTATION into the main road, greatly supports this fact.

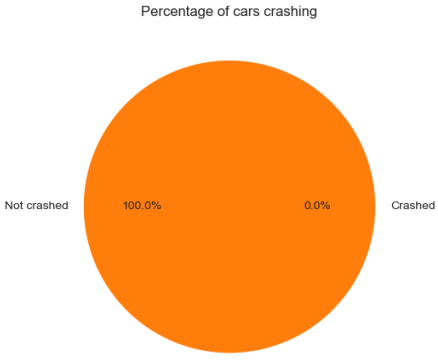

Figure 3: Short simulation collision ratio

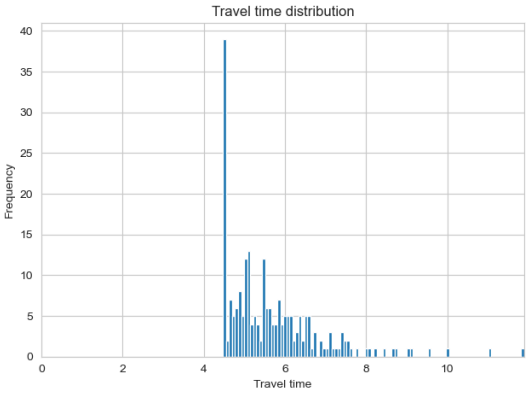

Figure 4: Short simulation CAR travel times

2.1.2. Long Simulation (B)

PARAMETERS:

Simulation length	GENERATOR limit	Runs generated
$t = 2000.0s$	$\tilde{l} = 100.0$	$n = 30$

We interpreted *long simulation* to mean a simulation where all CARS have the opportunity to leave the simulation. This way, there is plenty of time for the road to become filled with CARS, so that more complex inter-CAR interactions happen and may lead to crashes.

The simulation is run for a long time, so there is no opportunity for congestion. But since the ROADSTRETCH is a straight piece of road, and the main (non-GASSTATION) road never has to give priority to another merging roadway, the cars on the main road will still cruise along at a fast pace. This is reflected by the **towering peak of speedy arrivals** in Figure 6. The majority of the CARS can reach the collector fast.

This is likely aided by the relatively **fast maximum allowed velocity ($30\frac{m}{s}$)** on each ROADSEGMENT, compared to the **fairly short length of the main ROADSEGMENTS ($5m$)** that connect the GENERATOR with the COLLECTOR.

Once the slower and faster cars start to be mixed in the together, they start to impede one another. This will still result in slightly longer travel times for some cars. But the travel time graph Figure 6 does not show it due to the addition of cars that had to refuel.

The refuel time mean is 600s, with a s.d. of 130s. Since the simulation is 2000 seconds long, all of the refueling CARS get the chance to leave the simulation. This leads to the approximately normally distributed set of travel times around $t = 600$s

All of this results in a collision rate of 3.1%. The graph of collision locations Figure 7 shows in what DEVS components which percentage of the total amount of collisions took place. The components are named as follows:

- lower_seg_2: The segment that the GASSTATION outputs cars onto. This segment merges into seg_col_0
- seg_col_x: seg_col_0 is where the main GASSTATION output segment merges back into the main road. seg_col_9 is the final road segment before the collector. All others are connected in order.

We filtered out the segments where no collisions occurred. It is clear that no collisions occurred in the stretch of road leading from the generator to the GASSTATION. This is likely a result of the GENERATOR implementation. The GENERATOR, by design, only outputs a new CAR when the connected output segment is free. In addition to that, after outputting a CAR, the GENERATOR idles for IAT. Thus, when outputting a CAR, the previous CAR has plenty of time to get a fair distance away from the generator.

Oppositely to this, all cars need to go to the COLLECTOR. So faster cars have the opportunity to gain ground on slower cars over the course of the road stretch. This explains why most collisions occur on the road segments leading to the connector. In addition, the cars that leave the GASSTATION merge into col_seg_0.

In addition to this, many collisions occur is lower_seg2. This is the ROADSEGMENT leaving the GASSTATION. The GASSTATION implementation outputs a car that is ready to leave after waiting the `t_until_dep` time of lower_seg2. Since the main road has priority over lower_seg2, this means that oftentimes cars in lower_seg2 will be forced to slow down or wait. This results in the GASSTATION causing quite a few crashes.

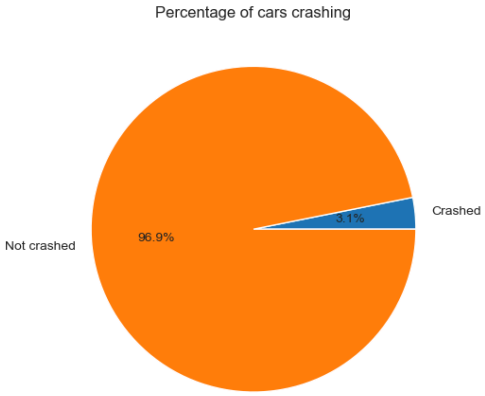

Figure 5: Long simulation collision ratio

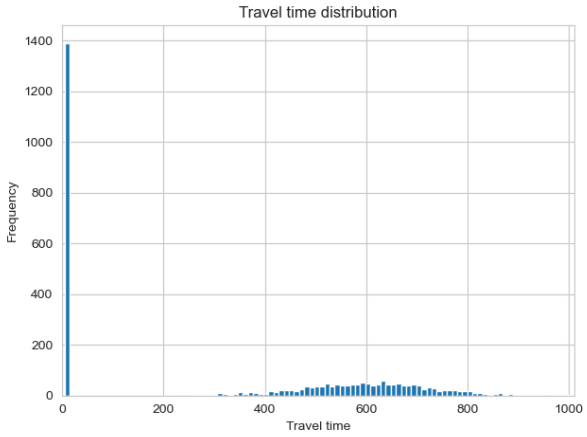

Figure 6: Long CAR travel times

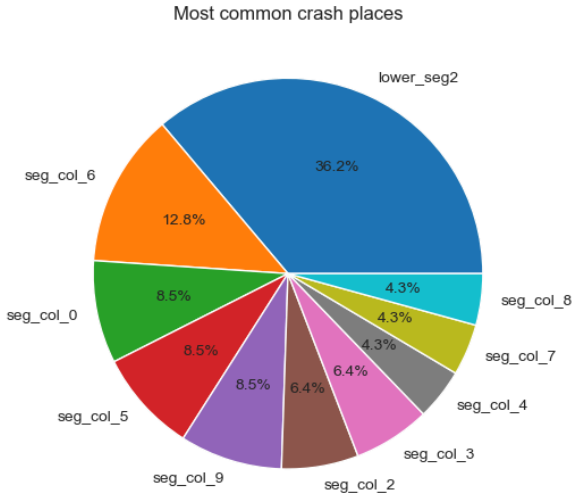

Figure 7: Long simulation collision locations

2.1.3. Observations (B-C)

It is clear that longer simulations result in more collisions due to interactions between cars on the main road and cars leaving the gasstation. The priority of the main road may cause collisions due to the GASSTATION.

We prefer longer simulation times. They will provide a clearer picture of how cars will behave in congested situations, and in when multiple merging roadsegments actually have cars interact and merge. They also give more accurate results with relation to the randomness of car velocities and IAT waiting times.

The travel times show that cars that do not refuel generally arrive very fast. The main road by itself causes fairly few collisions due to long IAT. The GASSTATION refuel delay results in a normally distributed group of arrival times around the refuel delay mean. The cars that do need to refuel by default spend much longer in the simulation. This could be mitigated by simulating using more ROADSEGMENTS between GENERATOR and COLLECTOR.

2.1.4. Order of Execution

The order of execution of Atomic DEVS depends on the coupled DEVS selection function. This function is a tie breaker when more than one event, internal or external, would occur on the same global time.

The default select function for the pypDEVS coupled DEVS models returns the first element of specified the list of options.

```
class CoupledDEVS(BaseDEVS):
    """
    Abstract base class for all coupled-DEVS descriptive classes.
    """

    # ....

    def select(self, imm_children):
        """
        DEFAULT select function, only used when using Classic DEVS simulation

        :param imm_children: list of all children that want to transition
        :returns: child -- a single child that is allowed to transition
        """
        return imm_children[0]
```

Codeblock 4: Coupled DEVS selector code (location: `pydevs/DEVS.py`)

The pypDEVS solver implementation contains orders the list of options alphabetically. As a result, the order of execution of the AtomicDEVS is in alphabetical order when there are conflicts. This is reflected in the **pypDEVS solver source code**, which calls the CoupledDEVS select function on a list sorted alphabetically, using a lambda key function that returns the full model name as sorting key.

```
chosen = model.select_hierarchy[level-1].select(sorted(colliding, key=lambda i:i.getModelFullName()))
```

Codeblock 5: Simulator default selection function (location: `pydevs/solver.py`)

For context, this line originates from the following larger Codeblock 6 fragment, also found in the same **documentation/source code**. The solver only resorts to the select function when a tie breaker is needed, namely `if len(imminent) > 1`.

```
if len(imminent) > 1:
    # Perform all selects
    imminent.sort(key=lambda i: i.getModelFullName())
    pending = imminent
    level = 1
    while len(pending) > 1:
        # Take the model each time, as we need to make sure that the selectHierarchy is valid everywhere
        model = pending[0]
        # Make a set first to remove duplicates
        colliding = list(set([m.select_hierarchy[level] for m in pending]))
        chosen = model.select_hierarchy[level-1].select(
                sorted(colliding, key=lambda i:i.getModelFullName()))
        pending = [m for m in pending
                   if m.select_hierarchy[level] == chosen]
        level += 1
    child = pending[0]
else:
    child = imminent[0]
```

Codeblock 6: Full solver code (location: `pydevs/solver.py`)

To change the order in which they are simulated, we can define a custom selection function for the (RoadStretch) coupled DEVS model.

This would be unlikely to alter the simulation results of the RoadStretch model much. The most pressing problem with the current simulation is choosing whether the Car on the main road may drive along, or if the merging Car on the lower_seg2 GasStation output road may merge instead. This would prevent some collisions.

2.2. 4-Way CrossRoads Simulation (III)

Similarly to the RoadStretch model described in the previous section, we will now construct a Coupled DEVS that will connect some set of Generators using a series of RoadSegments to a set of Collectors. In the center of our road network, a CrossRoads component will connect all the different branches to eachother.

2.2.1. Regular Intersection (A)

2.2.1.1. Construction

Figure 8 shows how the CrossRoad may be connect to four generator and collector branches. Implementation-wise, each of the segments of the different branches are just connected to one another *sequentially*. If so desired, the amount of branches and length of those branches can be altered to any number.

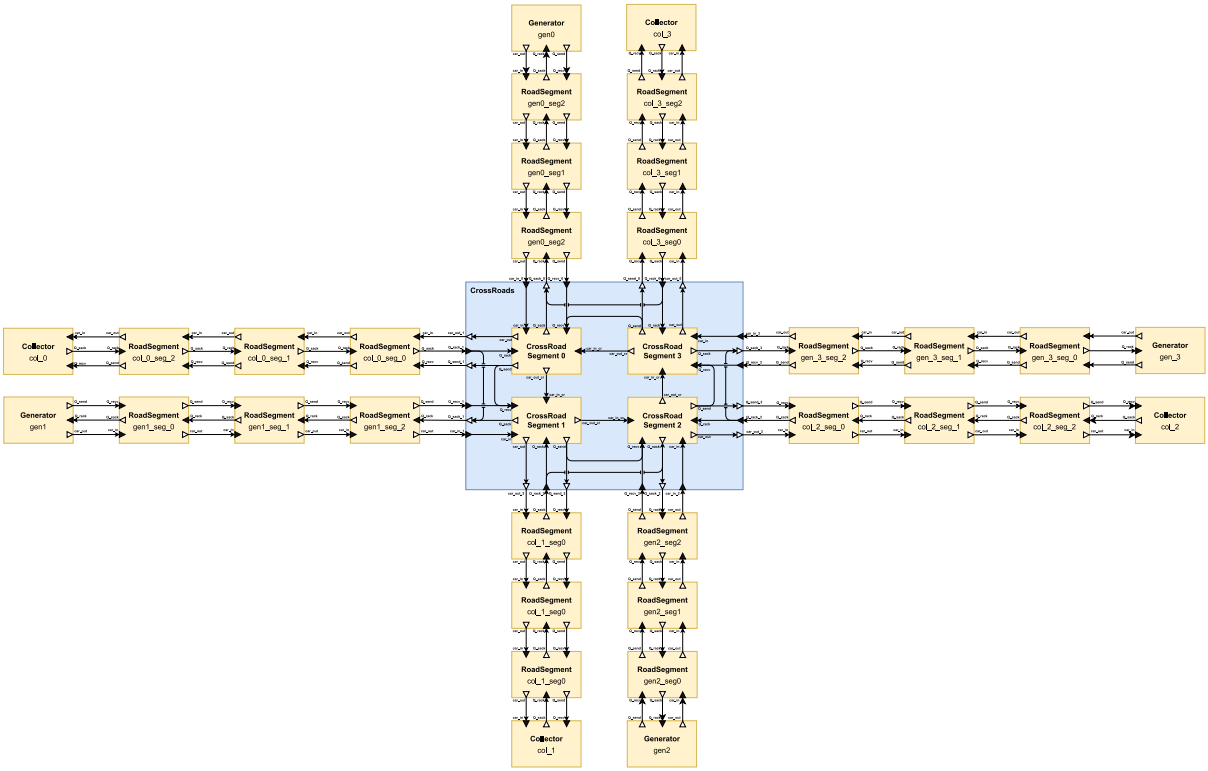

Figure 8: Standard 4-Way CrossRoads structure

2.2.1.2. Order of execution

After constructing a simulation framework for the CrossRoads (similarly to RoadStretch), we can inspect the internal behaviour of our CrossRoads components with respect to the larger model, and investigating when the segments get invoked in our simulator.

As with the RoadStretch model, we can again note that the ordering depends on the following:

1. The segment's `timeAdvance()` output is lowest
2. If the resuling timers are identical, then take the component first in the alphabetical ordering

This behaviour can easily be adapted by defining a `select()` method within the CrossRoads coupled DEVS. For example, if two CrossRoadSegments could be visited at the same time, we can choose to take the one with the lowest amount of car arrivals.

Depending on how the `select()` function is chosen, it either has no discernible influence (choosing at random), or slightly reduce the collision rate and travel times in the model. One example for the latter can be found by setting the order such that crossroad segments adjacent to empty crossroads segments are considered first.

2.2.2. Right-of-Way Intersection (B)

For the purposes of this assignment, the Right-of-Way traffic rule can best be stated as:

When two cars meet at an intersection,
*a car coming from the right — from the perspective of current car — has **priority**.*

In our previous example, an example of such a scenario can be found at CrossRoadSegment 1, where both RoadSegment *gen_1_seg_2* and CrossRoadSegment 0 meet. In this case, from the perspective of CrossRoadSegment 0 (who wants to go to the next adjacent segment), cars from *gen_1_seg_2* arrive from the *right*.

In other words: cars from *gen_1_seg_2* have priority over cars from CrossRoadSegment 0, or more generally, cars coming from previous CrossRoadSegments give way to cars entering the CrossRoads.

As seen in RoadStretch, we can apply the principle of a segment having priority over another, using a SideMarker, and three specific connections. Figure 9 shows how the priority segment, regular segment and SideMarker should be linked to one another. In general, it concerns following three connections:

- **regular$_{Q_{send}}$ → priority$_{Q_{recv}}$** (if a car arrives, request `time_until_dep` from priority seg.)
- **priority$_{Q_{sack}}$ → merge$_{mi}$** (after `observ_delay`, acknowledge the Query)
- **merge$_{mo}$ → regular$_{Q_{rack}}$** (mark the QueryAck as `sideways` and forward to regular seg.)

Note that for consistency and clarity, these colorings will be re-used for the full diagrams for Right-of-Way and Roundabout.

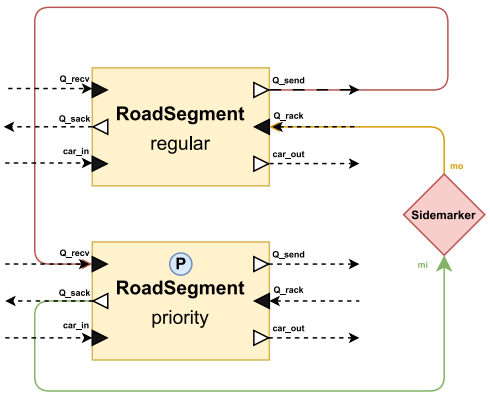

Figure 9: General priority segment structure

However, actually getting to link these ports to one another, proved a bit more difficult than initially thought. The 4-Way CrossRoads is implemented as a coupled DEVS, containing the CrossRoads as a component.

Due to DEVS restrictions, it is not possible to connect a port from our parent coupled DEVS (e.g. *gen_1_seg_2*.Q_{sack}) to a port of a component in a child coupled DEVS (e.g. *merge1*.mi). We bypassed this problem by introducing a new port to the CrossRoads called `mi_*`, which simply connects `mi_* -> merge_i.mi`. To implement the Right-of-Way priority in full, we had the following connections (also shown in Figure 10):

- **CrossRoads/Q_{send_i} → gen$_i$seg$_2$/Q_{recv}** (Q_{send_i} is connected to CrossRoads/segment$_{i-1}$/Q_{send})
- **gen$_i$seg$_2$/Q_{sack} → CrossRoads/mi$_i$**
- **CrossRoads/mi$_i$ → CrossRoads/merge$_i$/mo**
- **CrossRoads/merge$_i$/mo → CrossRoads/segment$_{i-1}$/Q_{rack}**

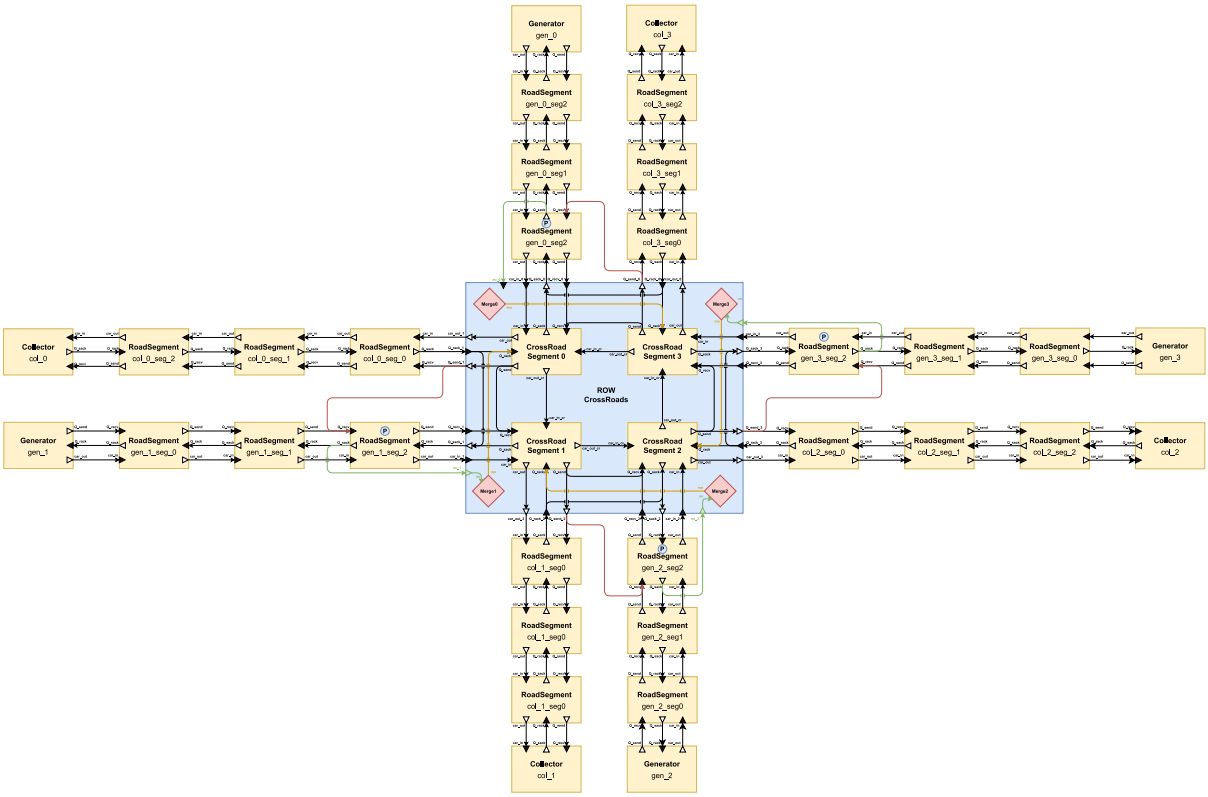

Figure 10: Right-of-Way 4-Way CrossRoads structure

However, simulating this model quickly shows that the model frequently ends in a deadlock (infinite Query polling with the cars never being able to move to either of the next segments). The deadlock problem is inherent from the design of the priority, and further compounded by another aspect of the Generator implementation.

The explanation for this behaviour starts from the important observation that a car at an input segment will **always** have priority for entering the next segment, over the previous CrossRoadSegment. In effect, the car on the previous segment may *not* move at all (neither moving towards the next segment, or exiting the CrossRoads core).

So when the CrossRoads core is saturated — meaning that every segment contains a car — then every adjacent CrossRoadSegment is blocked from moving. Since not a single segment may move, the cars in the core cannot cycle nor exit to free up a spot, and the cars at the inputs will keep blocking the movement.

A partial solution for this problem, is to *only* stall the car if its destination segment is currently occupied. A car at $\text{CrossRoadSegment}_0$ with destination Collector_0 should just be able to exit — regardless of whether or not the next segment is occupied, or a car is on priority-segment Gen_1Seg_2. However, in order to implement this, the QueryAck message would need to contain a **source** state, such that the speed updating function only needs to take the relevant occupancy messages into account. We *did* implement this field in QueryAck, but we are only using it for computing statistics; it is never used in receiveAck() functions.

As alluded to earlier, the problem is also exacerbated by the fact that our Generators will output the first cars at the same moment, meaning that they also arrive at the core at around the same time.

2.2.3. Roundabout Intersection (C)

Similar to the Right-of-Way CrossRoads, the roundabout will have a priority relationship between two different RoadSegments. The priority principle here can be described as:

When two cars meet at an intersection,
*a car already on the crossroads has **priority** over an car that wished to enter.*

In this case, then, we state that cars on the connecting input segments give way to cars on the adjacent CrossRoadSegment. Generally: CrossRoadSegment$_{i-1}$ $\rightarrow_{\text{priority}}$ InputSegment$_i$.

Practically, this means that the following edges should be connected:

- **CrossRoads/Q$_{\text{recv}i}$ → CrossRoads/segment$_{i-1}$/Q$_{\text{recv}}$** (Q$_{\text{recv}i}$ is connected to InputSeg$_i$/Q$_{\text{send}}$)
- **CrossRoads/segment$_{i-1}$/Q$_{\text{sack}}$ → CrossRoads/merge$_i$/mi**
- **CrossRoads/merge$_i$/mo → CrossRoads/Q$_{\text{sack}i}$** (Q$_{\text{sack}i}$ is connected to InputSeg$_i$/Q$_{\text{rack}}$)

Since all connections solely exist within CrossRoads, there was little difficulty in getting the ports connected properly. Note that in Figure 11, the connections between the different CrossRoadSegments and the CrossRoads input/output ports are hidden away, due to the generally messy nature of the added connections.

The Roundabout itself has no major issues inherent to its design, other than the deadlock issue it inherited from the base Free-for-All CrossRoads. Since access to the CrossRoads core is now strictly limited, the largest cause of worry are the longer waiting times for cars on InputSegments resulting in extensive traffic jams or even collisions due to car-backups.

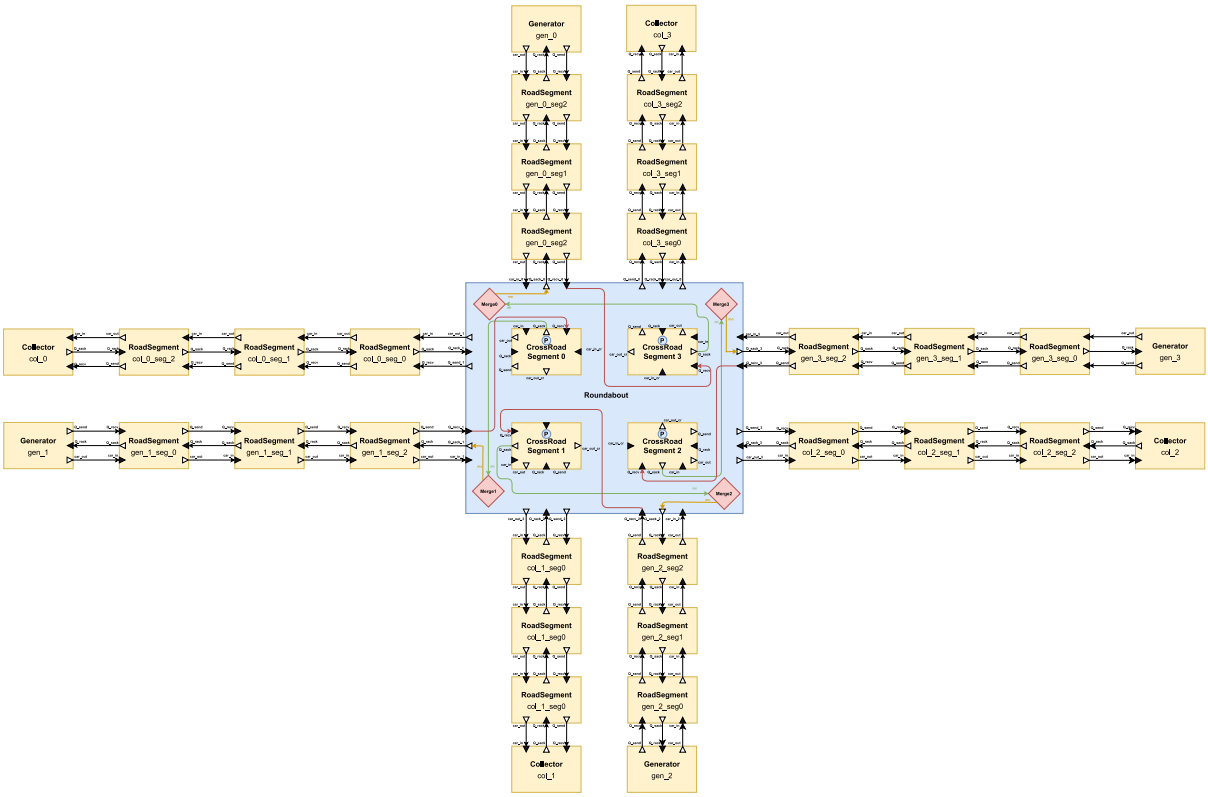

Figure 11: Roundabout 4-Way CrossRoads structure

2.2.4. Intersection Type Comparisons (D)

Here are some general observations for the different CrossRoads types:

- Roundabout and regular CrossRoads have generally the lowest collision percentage
- CrossRoads with priority (ROW and Roundabout) have longer travel times due to stalling
- In roundabout, most crashes happen *outside* the CrossRoads

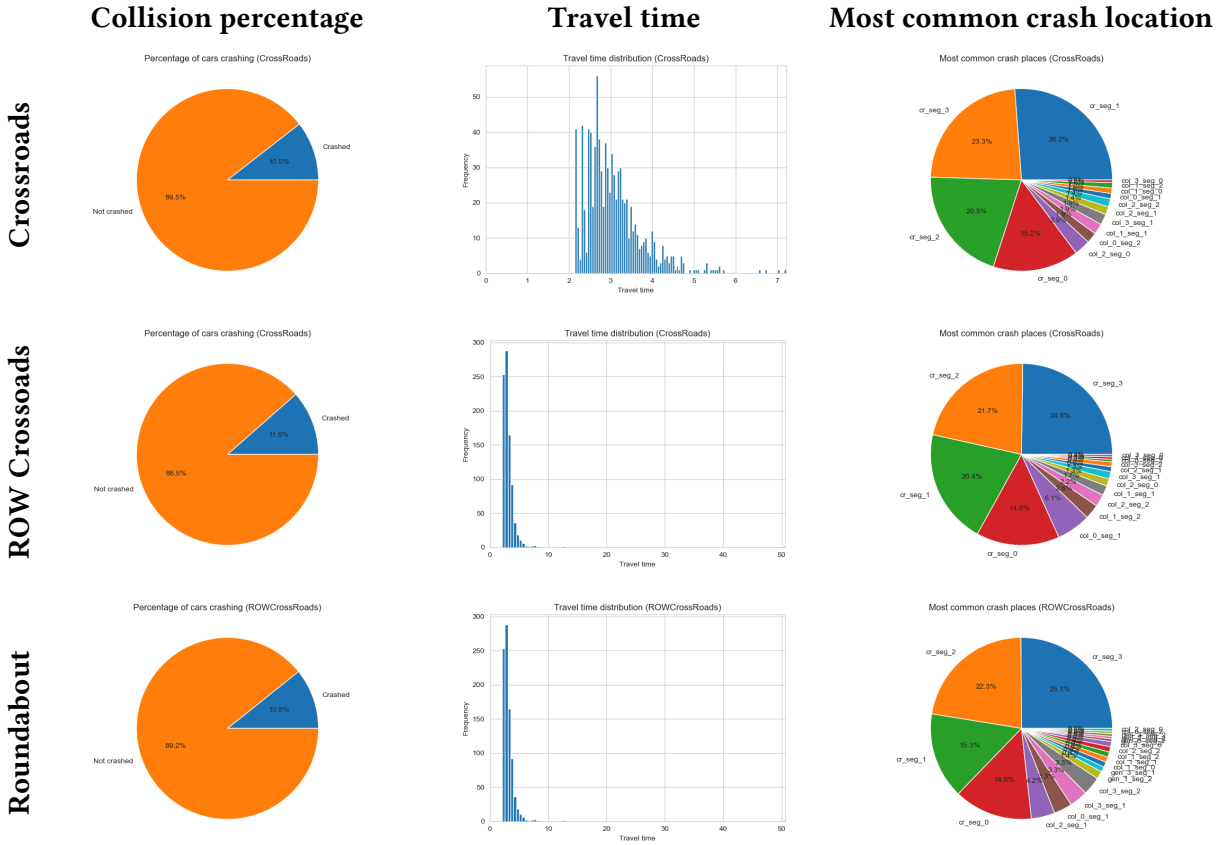

Figure 12: Overview of simulation results for different CrossRoads types