*Proceedings of the 2023 Winter Simulation Conference*
*C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.*

# CLAVS/ODVS: COMBINING CLASS/OBJECT DIAGRAMS AND DEVS

Jordan Parezys
Randy Paredis
Hans Vangheluwe

University of Antwerp – Flanders Make
Middelheimlaan 1, Antwerp BELGIUM

## ABSTRACT

The Discrete EVent system Specification (DEVS) formalism is a modular discrete-event modeling formalism. It has a formal specification in terms of systems theory and is supported by several efficient and usable simulator implementations. In these implementations, the DEVS formalism is often "grafted" onto an existing Object-Oriented programming language. Examples are C++ in the case of ADEVS and Python in the case of PythonPDEVS. To match this grafting, we present CLAVS, the CLAss diagram and deVS formalism and its instance counterpart ODVS, the Object Diagram and deVS formalism, and their visual notations. These languages use an automaton-like visual notation for Atomic DEVS models and a Class Diagram notation augmented with port information and event structure specification. An implementation of a visual CLAVS/ODVS modeling environment built on `draw.io` is presented. The use and usefulness of the formalism is demonstrated by means of a simple traffic model whose detailed specification is presented.

## 1 INTRODUCTION

System engineers *design*, *analyze*, and *deploy* complex, software-intensive, Cyber-Physical Systems (CPS). Multi-Paradigm Modeling (MPM) (Mosterman and Vangheluwe 2004), advocates explicitly modeling all relevant aspects of a system using the most appropriate modeling language(s), at the most appropriate level(s) of abstraction. DEVS (Chow and Zeigler 1994) is a popular modeling formalism that focuses on state changes triggered by *events*. It is a general-purpose discrete-event modeling language and may serve as an "*assembly language*" onto which other modeling languages can be mapped (Vangheluwe 2000). However, the most appropriate *notation* for DEVS, as used in practice, is by grafting it onto an Object-Oriented Programming (OOP) language. Examples are PythonPDEVS (Van Tendeloo and Vangheluwe 2016), DEVSJAVA (Sarjoughian and Zeigler 1998), adevs (Nutaro 2015), and Cadmium (Belloli et al. 2019). This "*grafted*" approach does not only allow simplified encoding of the state and functionality, but also allows instantiating (helper) classes to be used in the model. In essence, this yields *class diagrams* (Object Management Group 2002) that *represent* DEVS models.

It is pertinent that the resulting structure combining DEVS and Class Diagrams has a precise definition that is compliant with the DEVS specification (Li et al. 2011). Furthermore, it must be as implementation-language independent as possible. This implies that specific OOP language constructs should be made explicit to ensure the functionality persists in every OOP implementation (Barroca et al. 2015). When a visual notation is used, it should be user-friendly and meaningful in the context of DEVS modeling (Maleki et al. 2015). Additionally, *debugging* tool support is crucial for usability of modelling formalisms (Van Mierlo et al. 2017).

In this work, the CLAVS formalism is introduced as a combination of Class Diagrams and DEVS, supporting the previously listed features. During simulation, instances of DEVS (Class) models are created, evolve and interact, leading to behavior traces. ODVS formalism, a combination of Object Diagrams and DEVS is introduced to formalize and represent these instance-level artifacts.

**Structure.** Section 2 introduces in detail a simple, yet representative example from the traffic domain. In section 3, CLAVS and ODVS are introduced by means of the traffic example. Section 4 describes related work and section 5 concludes the paper.

## 2 RUNNING EXAMPLE

While the practicality of CLAVS has been explored in a project with an industrial partner, this paper focuses on a simple DEVS example from the traffic domain that covers all elements of the DEVS specification. This example has been used for the last two decades in courses at McGill University and the University of Antwerp to test students' understanding of the DEVS formalism.

To introduce CLAVS and ODVS, we use the simplest possible usable traffic model: car traffic on a straight stretch of road. This road is made up of a sequence of small road segments. Each road segment can hold exactly one car. If more than one car is present in a road segment, a collision occurs.

The model consists of a coupled model named `RoadStretch`, which is made up of a concatenation of one `Generator`, followed by a series of `RoadSegments`, and terminated by a `Collector`, as depicted in Figure 1.

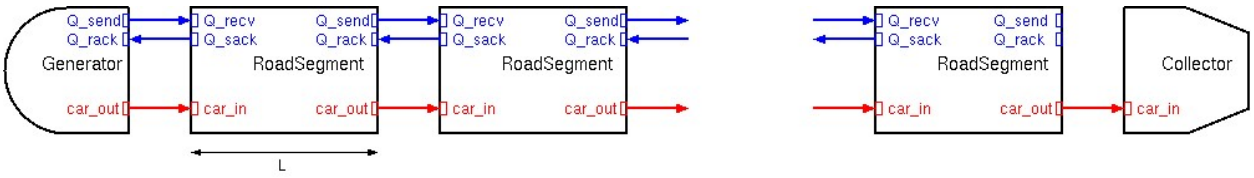



Figure 1: Road stretch use-case visualization.

The following sections describe all components and elements of this example.

### 2.1 Car

Cars are instances of the `Car` class and are generated by the `Generator` Atomic DEVS. They are passed through a sequence of `RoadSegment` Atomic DEVS components, to finally end up at the `Collector` Atomic DEVS.

The `Car` is a container for all the relevant information pertaining to a car (and its driver, if that level of detail is required), set at initialization time: (1) a unique `ID`; (2) the car(driver)'s preferred speed `v_pref` (*i.e.,* the speed that the car tries to attain); (3) the car's maximum acceleration (velocity increase) over one road segment (`dv_pos_max`); (4) the car's maximum deceleration (velocity decrease) over one road segment (`dv_neg_max`); and (5) the time at which the car leaves the `Generator` (`departure_time`). The attribute `v` keeps track of the car's current speed.

We also give the `Car` an attribute `distance_travelled`, which changes during the course of the simulation to reflect its changing state. The attribute `distance_travelled` is initialized to 0. Each time a `Car` object leaves a road segment, `distance_travelled` is incremented with `L`, the length of that road segment. Thus, at each point in simulated time, `distance_travelled` reflects the distance the car has travelled, irrespective of the traffic system's topology and the particular route taken by the car.

This entity is output on the `car_out` ports and input in the `car_in` ports from Figure 1.

### 2.2 Query and QueryAck

The moment a car enters a new road segment, a `Query` message is sent to the next road segment through the sender road segment's `Q_send` output port. The receiving road segment receives the `Query` through its `Q_recv` input port.

This Query is used to model, at a discrete event level of abstraction, the driver's observation of the next road segment for the presence of a car. In this style of discrete event modeling, where the road segments are the active components, the next road segment replies, after some observation delay observ_delay, with a QueryAck message through its Q_sack "query send acknowledgement" output port. The QueryAck message is received by the Query's sender on the latter's Q_rack "query receive acknowledgement" input port.

While the Query has no attributes, the QueryAck carries information about the presence of a car in the next road segment. It contains the single attribute t_until_dep, which can be 0 (no car is present in the next road segment), a strictly positive number (time delay until the car in the next road segment exits that segment, *i.e.,* the time until it is safe to enter the next road segment) or $+\infty$ (the car in the next segment has velocity 0, meaning a collision has occurred). A CarId attribute was added to ensure correct identification of the messages.

### 2.3 Generator

A Generator models the road system's "environment" by producing Cars entering the system. It generates Car instances on its car_out output port. The Inter-Arrival Time (IAT) of cars is uniformly distributed over the interval [IAT_min, IAT_max). Every time a Car instance is generated, it is passed a value for v_pref by the Generator. This value is sampled from a uniform distribution over the range [v_pref_min, v_pref_max).

Note that it is likely that at the moment of car departure, there is still a car in the first road segment, which will lead to a collision. The solution is for a Generator to have a Q_send output port and a Q_rack input port exactly like a RoadSegment. A Generator should, like a RoadSegment, look forward at the road segment ahead. If a collision could occur, the Generator delays producing the next car's arrival.

This implies that a car output should not be produced when the IAT elapses, but the generator rather needs to go into another mode in which it immediately (after time delay 0) produces a query output to check if there is a car in the first road segment. If we were to use a non-zero time-delay, that would bias our IAT distribution which does not reflect reality. The time it takes to receive an acknowledgement to the query is the observation delay. Note that after we output the query, we keep waiting (until we receive an acknowledgement) for a duration $\infty$, as we really need to know what lies ahead. Note that in practice, a Generator will never be immediately connected to a Collector.

When the reply received indicates there is no car ahead, we can output the car immediately. Note that receiving an acknowledgement triggers the external transition function. It is however only at the time of an internal transition that an output can be generated. So, we need to add another zero time delay after which a car should be output and a transition is made to the original mode, where we the system will stay for IAT time.

When the reply received indicates there is a car ahead, we need to wait. As for a regular road segment, we schedule our departure after a time given by the t_until_dep attribute of the reply received.

### 2.4 Collector

The Collector models the road system's "environment" by accepting Cars leaving the system. Its main purpose is to collect statistics. For this simple use-case, we are interested in the following two statistics: (1) the average transit_time of each car and (2) the average deviation between the car's average speed and the v_pref for each car.

Note how unlike a RoadSegment, a Collector does not have a Q_recv input port nor a Q_sack output port. This means that Query messages sent from the last RoadSegment do not go anywhere and hence that last RoadSegment never receives a QueryAck. This is reasonable as a Collector has an infinite capacity for collecting cars. The fact that the last RoadSegment never receives a QueryAck

is not a problem. A car entering that last road segment just continues at the `v_old` velocity at which it entered the segment. It is thus scheduled to leave that segment after `L / v_old`.

### 2.5 Road Segment

A road segment has the following parameters: (1) the length `L` of the road; (2) the maximum allowed speed `v_max`; and (3) `observ_delay`, the delay between obtaining a `Query` and sending a `QueryAck`.

The `RoadSegment` has a list `cars_present` as part of its state to keep track of the cars currently in that road segment.

The moment a car enters a road segment (at velocity `v_old`, found in the car's attribute `v`), that road segment first checks if there are already cars present. If there are, the arriving car is added to the `cars_present` list, and all cars' velocities `v` are set to 0, denoting a collision.

If there are no cars present however (the list of present cars is empty), the `RoadSegment` immediately sends a `Query` message to the model downstream, via the ouput port `Q_send`.

It also schedules a departure of the car at time `L / v_old`.

Note how there may only be one downstream model as otherwise there would be a choice of where to go. This should be modelled explicitly in a choice model (not included here for brevity).

Some elapsed time later, a `QueryAck` is received interrupting the scheduled departure. During that time, the car will have travelled a distance `xi = elapsed * v_old`. There still remains a distance `remaining_x = L - xi` to be travelled to leave the road segment.

When a `QueryAck` is received, the road segment's external transition handles it. It updates the distance still to be travelled, and retrieves from the `QueryAck` object, the `t_until_dep` of the car in the next road segment. This time is used locally as `t_no_coll`, the minimum time the car must stay in the current road segment to not collide with the car in the next road segment when leaving. Note how collision may still occur as (1) the `t_until_dep` received in the `QueryAck` is only an approximation and (2) the car may not be able to slow down sufficiently within the remaining distance in the current road segment.

```
function calculateCarSpeed() {
    xi = elapsed * v_old
    remaining_x = L - xi
    t_no_coll = ack.t_until_dep
    v_new = remaining_x / min(v_pref, v_max)
    v_new = remaining_x / max(t_no_coll, v_new)
    return min(v_old + dv_pos_max, max(v_old - dv_neg_max, v_new))
}
```

Let us first consider the case where there is no car in the next road segment and the `t_until_dep` received in the `QueryAck` is thus 0. This case is depicted in Figure 2 in two examples.

As there are no restrictions on the speed imposed by the next road segment, the car will want to update its speed to its `v_pref`. However, the car should not exceed the current road segment's speed limit `v_max`. The new target speed is thus `min(v_pref, v_max)`. It may not be possible to attain this target speed due to the maximum velocity changes `dv_pos_max` and `dv_neg_max`. The final new speed `v_new` takes this into account. A departure is scheduled after `t_until_dep = remaining_x / v_new`.

Let us now consider the case where there is a car in the next road segment and the `t_until_dep` received in the `QueryAck` is thus a positive number. This case is depicted in Figure 3 in two examples.

Now, we cannot just set the target speed to `min(v_pref, v_max)` as that might bring us to the next road segment too early (i.e., before the car in that segment has left). The time until departure must thus be the maximum of `t_no_coll` (obtained from the `QueryAck`'s `t_until_dep`) and the time it would take to leave at the intended target speed `remaining_x / min(v_pref, v_max)`. This requires an updated speed of `remaining_x / max(t_no_coll, remaining_x / min(v_pref, v_max))`. It may not be possible to reach this speed however due to the maximum velocity changes `dv_pos_max`

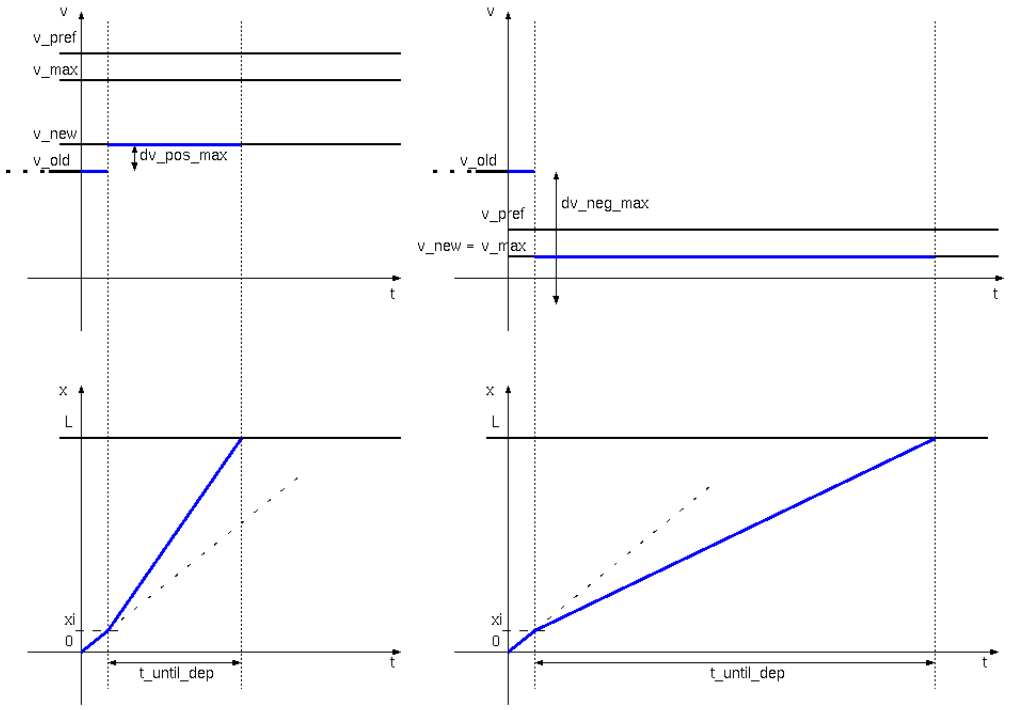



Figure 2: Two examples of adapting the speed of a car w.r.t. an empty road segment ahead.

and `dv_neg_max`. The final new speed `v_new` takes this into account. A departure is scheduled after `t_until_dep = remaining_x / v_new`.

An extension to the use-case can be made by adding a crossroad (intersection) segment component which inherits from the standard road segment. Instead of only allowing through traffic, the crossroad segment also allows exiting in another direction. Multiple crossroad segments can be combined to create any n-way-intersection or roundabout. An additional input and output port need to be added, indicating whether traffic to this segment came from/goes to the intersection itself. Though this and a further extension for signals/traffic lights are needed to model realistic traffic networks, it does not add to our explanation of the CLAVS and ODVS formalisms. These segment types will hence not be discussed.

## 3 CLAVS AND ODVS

The use-case will be modeled using the CLAVS and ODVS formalisms. CLAVS is composed of Class diagrams and DEVS.

### 3.1 Class Diagrams

Class diagrams (Object Management Group 2002) are used to represent the internal class-structure of software. It consists of *class signatures* (depicted by boxes) and *associations* (depicted by lines). Each class signature lists a set of attributes and methods that belong to instances of that class. At the association end, a *multiplicity* may be given to specify how many class instances (objects) may be in a relationship. Hollow arrows denote inheritance, a re-use mechanism. Figure 4 shows a class diagram for the use-case.

The classes `Car`, `Query` and `QueryAck` represent (passive, in that they do not have behavior) events, while the others represent the active building blocks for the use-case model.

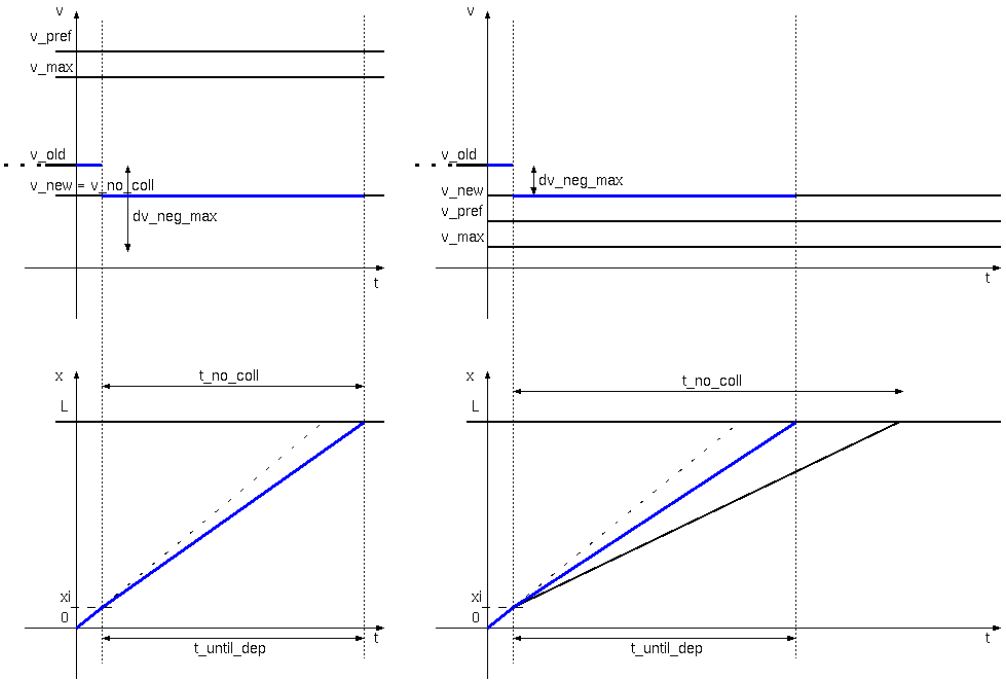



Figure 3: Two examples of adapting the speed of a car w.r.t. the road segment ahead.

### 3.2 DEVS

Discrete EVent System Specification (DEVS) (Chow and Zeigler 1994) is a block-based modeling language for precisely defining the behavior of discrete event systems. The blocks exchange *events*. An event can be modeled as a class that only has attributes. Each block has a *state* that may change throughout the simulation.

*Atomic DEVS* models are the most basic blocks in a DEVS model. They are described by an 8-tuple $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$. Here, $X$ defines the *input set* (*i.e.,* the set of accepted input events), $Y$ the *output set* (*i.e.,* the set of possible generated output events) and $S$ the *set of sequential states* of the model. The *internal transition function* $\delta_{int} : S \to S$ denotes internal state changes that result in an output event. Output events are produced by the *output function* $\lambda : S \to Y$. The *external transition function* $\delta_{ext} : Q \times X \to S$, with the *set of total states* $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$, produces a state change upon receiving an input event. $q_{init}$ denotes the *initial total state* ($q_{init} \in Q$) and $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ (*i.e.,* the *time advance function*) defines, in the absence of external input events, the time spent in a state before $\delta_{int}$ is invoked.

Atomic DEVS models may be visually represented as a Timed Finite State Automaton (TFSA). A TFSA consists of multiple states or modes (depicted by roundtangles) and transitions between them (depicted by arrows). Along the arrow is shown when a transition must be followed (either after a delay, or when an input is received).

In Figures 5, 6 and 7, the Atomic DEVS models for the `Generator`, the `Collector` and the `RoadSegment` are shown (respectively). The purple rectangle on the left depicts the state of the model. The inputs are depicted by the blue triangles on the left and the outputs are the red triangles on the right. Both the inputs and the outputs are *ports* that can only send events of a specific type. In the middle, the TFSA that represents the model's behavior is shown. Blue arrows depict external transitions and red arrows depict internal transitions. The labels on the arrows provide additional information. For external transitions, they start with the description of the received event. For internal events, a forward slash is used to denote which output is generated. All other information describes the internal logic executed when that arrow/transition is followed. For readability, complicated logic is encapsulated in helper functions.

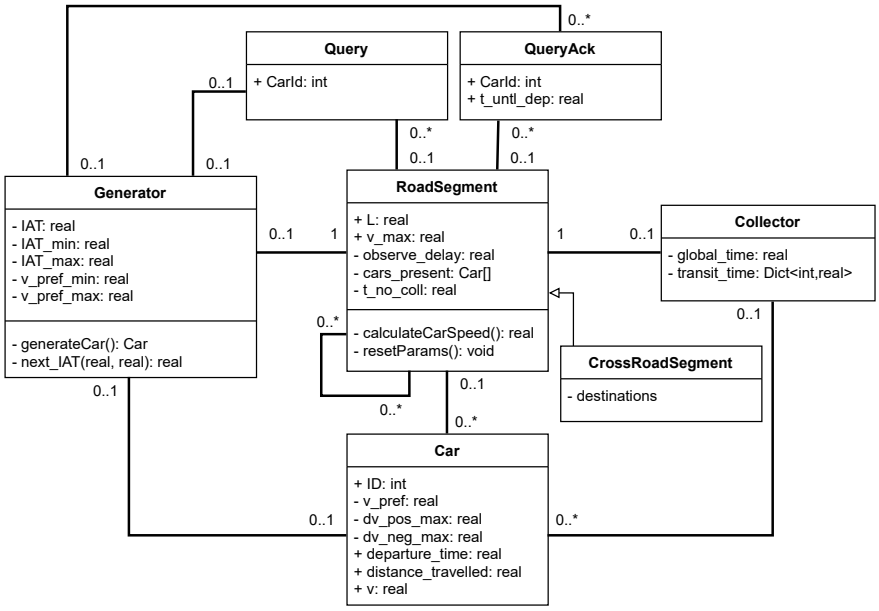



Figure 4: Road stretch class diagram.

The `Generator` (Figure 5) consists of four TFSA states, starting in "*Check Next Segment*", where a `Query` was sent to the next road segment. Upon receiving a `QueryAck`, a car is generated ("*Generate Car*") and the IAT is waited before outputting the car ("*Await IAT*").

The internal TFSA of the `Collector` (Figure 6) only contains a single state ("*Await Car*") in which it waits for a new car to exit a road segment. By storing all information about the cars in the state, the statistics can be computed at the end of the simulation.

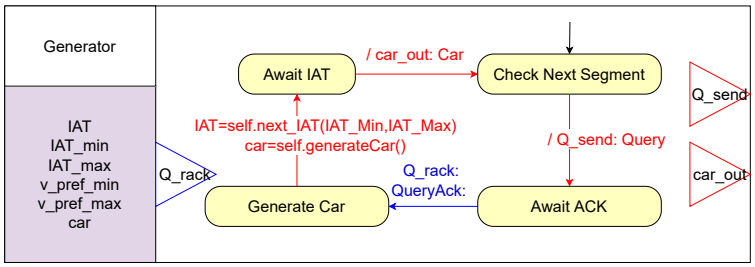



Figure 5: Atomic DEVS visualisation for the `Generator`.

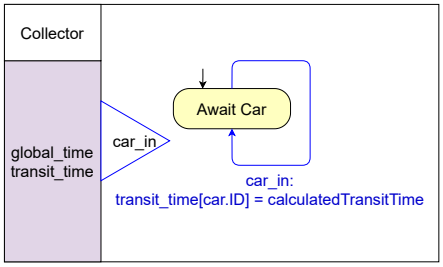



Figure 6: Atomic DEVS visualisation for the `Collector`.

The `RoadSegment` (Figure 7) contains six TFSA states and starts in "*Await Car*". Upon the arrival of a car ("*Receive Car*"), a `Query` is sent ("*Check Next Segment*"). If the `QueryAck` is received, the road segment becomes "*Occupied*" until `t_no_coll` time has passed, after which it returns to the initial state. When a new car arrives in an "*Occupied*" road segment, a "*Collision*" has occurred.

Multiple DEVS models connected in a network form a *Coupled DEVS* model. This network has structure $\Delta = \langle X_\Delta, Y_\Delta, D, M_i, I_i, Z_{i,j}, select \rangle$. Just like Atomic DEVS, $X_\Delta$ and $Y_\Delta$ denote the input and output sets of the network. $D$ is the *set of component references* and $M_i$ refers to the *model component* $i$, with $i \in D$). The *set of influencees* of component $i$ is denoted by $I_i, \forall i \in D \cup \{\Delta\}$ and $i \notin I_i$. The *transfer function* $Z_{i,j}$ ($Z_{\Delta,j} : X_\Delta \to X_j$; $Z_{i,\Delta} : Y_i \to Y_\Delta$; $Z_{i,j} : Y_i \to X_j$; $\forall i \in D \cup \{\Delta\}, j \in I_i$) allows for translating events from model output to model input along a connection. For instance, a `CarDeparture` event must become a `CarArrival` event. It is possible to make this translation implicit by allowing `CarDeparture` and `CarArrival` to be of the same class `Car`. Finally, the function $select : 2^D \to D$ allows for *tie-breaking*

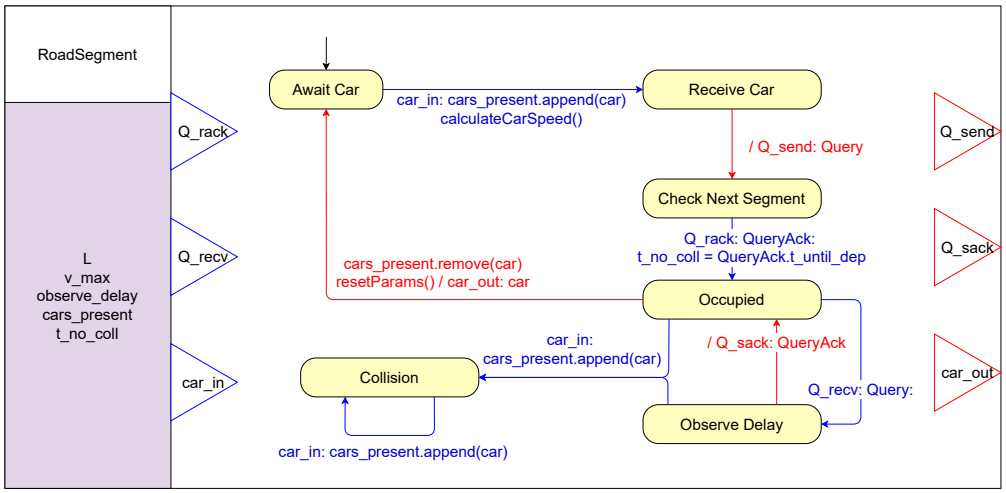



Figure 7: Atomic DEVS visualisation for the `RoadSegment`. Given the inheritance of the `CrossRoadSegment`, it also describes most of that component's logic.

when multiple $i \in D$ are due to transition at the same time. Often, the *select* function implements some form of priority scheme. Because a coupled model can be substituted by an equivalent Atomic DEVS model (*i.e.,* thanks to the *closure under coupling* of the DEVS formalism), arbitrary hierarchical models can be built.

Figure 8 shows a full CLAVS model of the Road Stretch example. As can be seen in the figure, this extends the class diagram from Figure 4 with additional DEVS-specific annotations. The ports have been added to the class descriptions, as well as a number in the top right to specify the multiplicity of the class instances. To indicate the set of allowed inputs on a port, the port names are amended with a comma-separated list of all allowed inputs. For the presented example, only a single event type is allowed for each port. Event classes have been given a red border. The red, striped arrow represents the DEVS connection between the multiple classes. The hollow arrow again depicts inheritance. All ports and fields are inherited, including the connections of the superclass. New members and ports have been added to the child.

### 3.3 ODVS

The instance-level complement of CLAVS are Object Diagrams + DEVS. An example is shown in Figure 9.

### 3.4 Graphical Notations

In the past, different graphical notations for DEVS have been developed. (Traoré 2009) bases itself on the concept of flowcharts and a so-called state event chart, mainly focusing on the event-based behavior of the individual DEVS components. It uses its own state notation, yielding a slight overhead to new users already familiar with FSAs. Like CLAVS, CD++ (Wainer et al. 2001) also allows modeling of DEVS components by means of simple state machines. Creating coupled models allows any components to be connected. CLAVS diagrams allow explicit specification of the allowed connections in a DEVS network. Maleki et al. (2015) introduced *DesignDEVS*, a conceptual method of modeling DEVS, also allowing Lua code (instead of just state machines) for the behavior of Atomic DEVS. The main idea behind DesignDEVS is that non-modeling experts can easily learn it.

Another graphical representation for DEVS can be obtained via the mapping from existing languages, giving them behavioral expressiveness. This has been done for SysML (Kapos et al. 2014; Nikolaidou et al. 2015) and AADL (Ahmad and Sarjoughian 2023) among others. A downside of those mappings is the lack of uniformity, platform-independence and tool support.

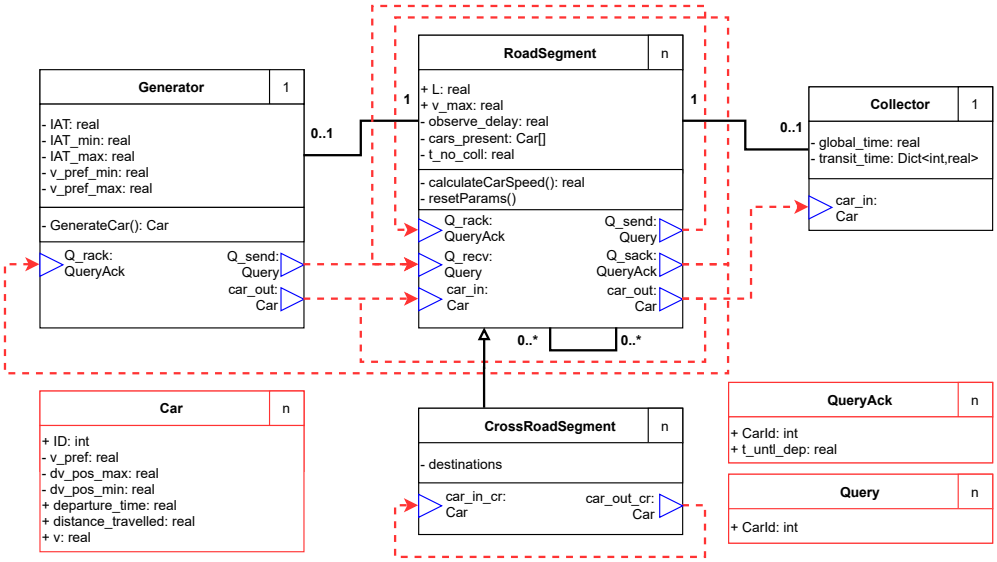



Figure 8: Road stretch CLAVS diagram.

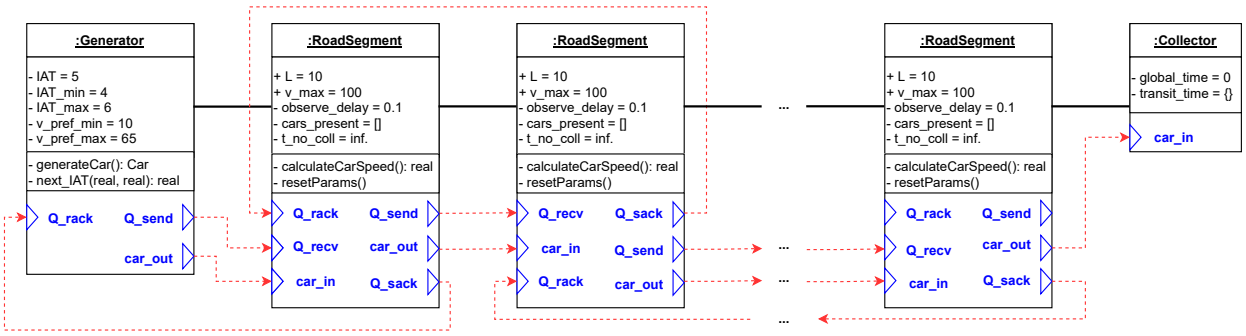



Figure 9: Road stretch ODVS diagram.

### 3.5 Tooling

In the past, multiple domain-specific (meta-)modeling environments were built. AToM$^3$ (de Lara and Vangheluwe 2002) made use of the primitive Python TkInter library whereas its successor AToMPM (Syriani et al. 2013) used the Raphaël Javascript library. `draw.io` (https://diagrams.net) is a graph-based diagramming tool that closely resembles both AToM$^3$ and AToMPM in terms of the creation of diagrams. Very little effort was required to use it for CLAVS and ODVS modeling, whilst maintaining builtin features such as sophisticated diagramming and SVG exporting.

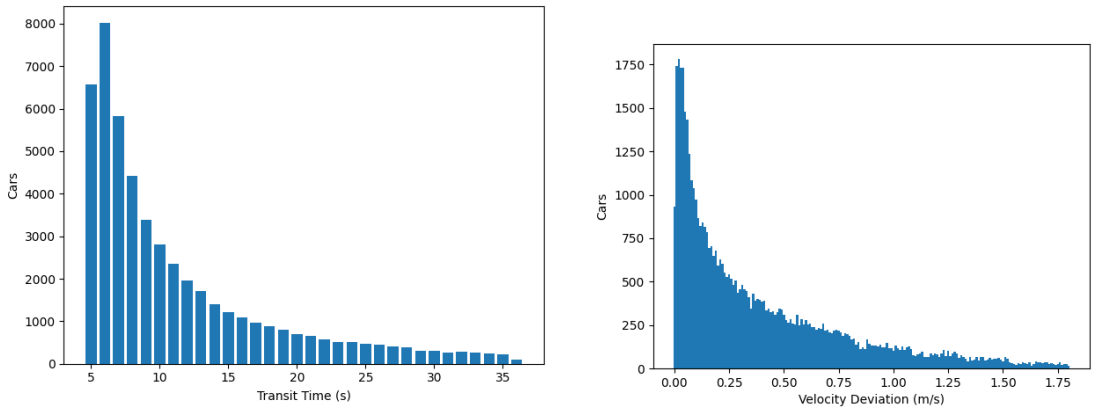
Additional advantages of `draw.io` are the usage of the *Electron 9* framework to allow both web-based and stand-alone use. The *Mathematical Typesetting* feature allows for rendering LaTeXformulas.

In `draw.io`, everything that is drawn is a "*shape*", which can have user-defined properties in the form of key-value pairs. A property `propName` can be used literally, or its value can be rendered using placeholders (*i.e.*, `%propName%`). Every time the value is changed, the diagram will be re-rendered. This feature is a direct result of the elegant and modular underlying *mxGraph* data structures (https://jgraph.github.io/mxgraph/) which keep the essential graph structure consistent with the visual graph information. The interactive behavior of `draw.io` can be customized by specializing callbacks. This allows for the inclusion of well-formedness checks written in JavaScript.

### 3.6 Simulation Results

When simulating this use-case with parameters chosen to reflect a busy highway, the transit time distribution is as shown in Figure 10a. This experiment setup assumes high velocities and low IAT. The first few cars complete the full trajectory fast, but as soon as some congestion starts to appear, the transit time gradually decreases.

The average deviation (in absolute value) between the actual velocity and the preferred velocity is shown in Figure 10b.




(a) Road stretch transit time distribution.

(b) Road stretch velocity deviation distribution.

Figure 10: Road stretch experiment results.

### 3.7 Implementation

CLAVS was designed with a few desired features in mind. First, a precise definition (compliant with DEVS) was required. The Atomic DEVS representation, as shown in Figures 5, 6 and 7, clearly incorporates the full 8-tuple of an Atomic DEVS. Second, CLAVS had to be as implementation-language independent as possible. While our Proof of Concept implementation uses PythonPDEVS, the CLAVS diagram itself was created using the target simulation language-neutral `draw.io` (http://diagrams.net). A library containing CLAVS components was created, allowing custom constraints on models. Connections between port of building blocks is built into the diagram editor `draw.io`.

A conversion script was designed to generate Python files compliant with PythonPDEVS. The action language used in the Atomic DEVS TFSA is still Python code. In order to be fully target simulation language agnostic, a neutral action language such as *DEVSLang* (Barroca et al. 2015) should be used instead. An advantage of a neutral language/framework is that the conversion to an OOP language may include several validation checks to ensure the end result is compliant with the DEVS specification. No usability studies were performed. However, CLAVS has successfully been used to solve problems with industry partners. Usability will be drastically improved once debugging is supported.

## 4 RELATED WORK

This work follows Multi-Paradigm modeling (MPM) (Mosterman and Vangheluwe 2004) principles. MPM advocates modeling all relevant parts and aspects of a system at the most appropriate level(s) of abstraction, using the most appropriate modeling language(s).

Li et al. (2011) does an analysis of multiple DEVS frameworks in OOP languages to verify the compliance with the original DEVS formalism, as was described in Chow and Zeigler (1994). In (Barroca et al. 2015), the usage and importance of a neutral action language is discussed, to make models independent

of the target implementation language. Song (2006) created a neutral modeling language and environment for DEVS, whereas Muzy and Nutaro (2005) describes an abstract simulator for DEVS. Frameworks such as PythonPDEVS (Van Tendeloo and Vangheluwe 2016), DEVSJAVA (Sarjoughian and Zeigler 1998), adevs (Nutaro 2015), Cadmium (Belloli et al. 2019) and many others implement such a simulator, albeit grafted on an OOP language.

DEVSJava, DEVSimPy (Capocchi et al. 2011) and DesignDEVS (Goldstein et al. 2016) are (conceptual) visual modeling environments for DEVS. Maleki et al. (2015) studied how to visualize DEVS modeling frameworks to allow non-programmers to easily understand the formalism.

Kapos et al. (2014) introduces a translation of SysML onto DEVS, giving an explicit execution semantics to an architecture model. While theoretically relatively close to CLAVS, the paper mainly encodes all DEVS logic in SysML, instead of co-existing.

## 5 CONCLUSION AND FUTURE WORK

We introduced CLAVS, the CLAss diagram and deVS formalism as well as its instance counterpart ODVS, the Object Diagram and deVS formalism, as well as their visual notations, by means of a simple traffic model whose detailed specification was presented. CLAVS uses an automaton-like visual notation for Atomic DEVS models and a Class Diagram notation augmented with port information and event structure specification. An implementation of a visual CLAVS/ODVS modeling environment built on `draw.io` was presented.

While CLAVS aims at being neutral in terms of OOP, the action (programming) language (as used to specify $\delta_{int}$ and $\delta_{ext}$ is still Python. We plan to use the implementation-language independent action language DEVSLang (Barroca et al. 2015) in the future.

Any formalism is always part of a family of complementary formalisms: modeling language, state language, trace language, and property language (Meyers et al. 2020). This implies that CLAVS/ODVS must be complemented by a Discrete Event State and Event Trace (DESET) formalism. This will be added, based on the trace language developed in Song (2006).

To be usable, modeling and simulation tools must support debugging. We plan to use our experience building a Parallel DEVS debugger in our meta-modelling tool AToMPM (Van Mierlo et al. 2017) to add debugging support to our Proof of Concept implementation of CLAVS/ODVS tooling.

In this paper, we have used logical arguments to convince the reader of the usefulness of the CLAVS/ODVS formalisms. A thorough empirical usability study is however still required.

## ACKNOWLEDGEMENT

This work was partially supported by the Flanders Make strategic research center.

## REFERENCES

Ahmad, E., and H. S. Sarjoughian. 2023. “An Environment for Developing Simulatable AADL-DEVS Models”. *Simulation Modelling Practice and Theory* 123:102690.

Barroca, B., S. Mustafiz, S. Van Mierlo, and H. Vangheluwe. 2015. “Integrating a Neutral Action Language in a DEVS Modelling Environment”. In *Proceedings of the eighth International Conference on Simulation Tools and Techniques*, SIMUTools ’15, 19–28: ICST.

Belloli, L., D. Vicino, C. Ruiz-Martín, and G. Wainer. 2019. “Building DEVS Models with the Cadmium tool”. In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 45–59. IEEE.

Capocchi, L., J. F. Santucci, B. Poggi, and C. Nicolai. 2011. “DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems”. In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 170–175. IEEE.

Chow, A. C. H., and B. P. Zeigler. 1994. “Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism”. In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, 716–722: IEEE.

de Lara, J., and H. Vangheluwe. 2002. "AToM$^3$: A Tool for Multi-Formalism and Meta-Modelling". In *FASE*, 174–188.

Goldstein, R., S. Breslav, and A. Khan. 2016. "DesignDEVS: Reinforcing Theoretical Principles in a Practical and Lightweight Simulation Environment". In *Proceedings of the 2016 Spring Simulation Multiconference*, 1–8. Pasadena, CA, USA: Society for Computer Simulation International (SCS).

Kapos, G.-D., V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos. 2014. "An Integrated Framework for Automated Simulation of SysML Models Using DEVS". *SIMULATION* 90(6):717–744.

Li, X., H. Vangheluwe, Y. Lei, and W. Wang. 2011, April. "A Testing Framework for DEVS Formalism Implementation". In *Proceedings of the 2011 Spring Simulation Multiconference*, 183–188. Boston, MA, USA: Society for Computer Simulation International (SCS).

Maleki, M., R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. 2015. "Designing DEVS Visual Interfaces for End-User Programmers". *SIMULATION* 91(8):715–734.

Meyers, B., H. Vangheluwe, J. Denil, and R. Salay. 2020. "A Framework for Temporal Verification Support in Domain-Specific Modelling". *IEEE Transactions on Software Engineering (TSE)* 46(4):362 – 404.

Mosterman, P. J., and H. Vangheluwe. 2004, September. "Computer Automated Multi-Paradigm Modeling: An Introduction". *SIMULATION* 80(9):433–450.

Muzy, A., and J. J. Nutaro. 2005. "Algorithms for Efficient Implementations of the DEVS & DSDEVS Abstract Simulators". In *1st Open International Conference on Modeling and Simulation*, 273–279. Clermont-Ferrand, France.

Nikolaidou, M., G.-D. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos. 2015. "Simulating SysML models: Overview and challenges". In *2015 10th System of Systems Engineering Conference (SoSE)*, 328–333. IEEE.

Nutaro, James J. 2015. "adevs". http://www.ornl.gov/~1qn/adevs/. Accessed 22$^{nd}$ April 2022.

Object Management Group 2002. "UML". http://www.uml.org/. Accessed 22$^{nd}$ April 2022.

Sarjoughian, H. S., and B. R. Zeigler. 1998. "DEVSJAVA: Basis for a DEVS-Based Collaborative M&S Environment". *SIMULATION* 30:29–36.

Song, H. 2006. "Infrastructure for DEVS Modelling and Experimentation". Master's thesis, School of Computer Science, McGill University.

Syriani, E., H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. "AToMPM: A Web-based Modeling Environment". In *Proceedings of MODELS'13 Demonstration Session*, 21–25.

Traoré, M. K. 2009. "A Graphical Notation for DEVS". In *Proceedings of the 2009 Spring Simulation Multiconference*, 1–7. San Diego, CA, USA: Society for Computer Simulation International (SCS).

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2017. "Debugging Parallel DEVS". *SIMULATION* 93(4):285–306.

Van Tendeloo, Y., and H. Vangheluwe. 2016. "An Overview of PythonPDEVS". In *JDF 2016*, 59–66: Cépaduès.

Vangheluwe, H. 2000. *Multi-Formalism Modelling and Simulation*. Ph. D. thesis, Universiteit Gent.

Wainer, G., G. Christen, and A. Dobniewski. 2001. "Defining DEVS Models with the CD++ Toolkit". In *Proceedings of ESS*, 633–637.

## AUTHOR BIOGRAPHIES

**JORDAN PAREZYS** developed CLAVS as part of his Master's thesis in the Modelling, Simulation and Design Lab (MSDL) at the university of Antwerp (Belgium). His e-mail address is jordan.parezys@hotmail.com.

**RANDY PAREDIS** is a Ph.D. student in MSDL. He develops a generic framework for model-based design of Digital Twins. He also develops techniques and tools to use DEVS as a common denominator for discrete-event and hybrid modeling languages. His e-mail address is randy.paredis@uantwerpen.be.

**HANS VANGHELUWE** is a Professor and head of the MSDL. He develops modeling and simulation methods, techniques and tools to increase system builders' productivity. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results. His e-mail address is hans.vangheluwe@uantwerpen.be.