Model-based Animation of Micro-Traffic Simulation – WIP

Philip Guin and Eugene Syriani
University of Alabama
paguin@crimson.ua.edu, esyriani@cs.ua.edu

Keywords: domain-specific model, DEVS, animation

Abstract

Testing discrete-event simulation models is often done at the level of traces. However, visual feed-back to the modeler is crucial as he is very often a domain expert with limited computing knowledge. We propose a generic framework for graphically animating the visualization of DEVS simulations to enable testing of the simulation model at the domain-specific level. This paper reports initial work for animating micro-traffic simulation with our framework.

1. INTRODUCTION

Micro-traffic simulation has been studied for many decades. Transportation engineers, urbanists, and governing authorities typically use it to investigate present traffic networks and evaluate projections. Visualization of micro-traffic network simulation is of greater importance for these users who are typically not computer scientists or simulation tool builders. A plethora of commercial and open-source visual traffic simulation tools exist [1], *e.g.,* CORSIM, NETSIM, SUMO, VISSIM etc. Although they are powerful tools to animate the simulation of traffic networks with lane changing, intersections and freeways, it is hard to incorporate conception of factors such as driver perception, reaction time, or recklessness. The traffic modeler would have to explicitly implement such features as plug-ins, in addition to modifying the kernel of the tool itself; these are certainly not trivial tasks for traffic modelers. Therefore, to further reduce the gap between modeler and programmer, our solution is a domain-specific modeling platform that generates individualized traffic simulators through model-driven techniques [2].

The ultimate goal of the research we propose is to enable testing of simulation models at the model level rather than at the level of simulation traces. In this context, the modeler defines a graphical domain-specific language (DSL), for example a language for modeling traffic networks. The behavior of the individual components of the DSL (*e.g.,* cars, roads stretches, intersections, traffic lights) is defined modularly in a given simulation protocol. In our case, we use the discrete-event system specification formalism (DEVS) [3]. The domain-specific model (DSM) is then automatically synthesized into a corresponding DEVS model. Then, the simulation of the DEVS model is recorded and re-played for animation purposes. In this paper, we describe the initial set up of the framework for animating the DSM given a simulation run.

In Section 2., we outline the generic architecture of our simulation animation framework. In Section 3., we describe our traffic DSL example and explain the implementation details of the animation in Section 4.. Finally we outline some relevant related work in Section 5. and conclude in Section 6..

2. ANIMATION ARCHITECTURE

As identified in [4], model-driven simulation requires a domain-specific expert (modeler) and a simulation expert. Our simulation animation framework relies on two technologies: AToMPM for graphical modeling and PythonDEVS [5] for simulation. On the one hand, the modeler defines a domain-specific language for modeling traffic networks. The modeling tool used in our framework is AToMPM [6], an online tool for designing domain-specific modeling environments and transform models. Clients connect to and communicate with the AToMPM server through HTTP requests. On the other hand, the simulation expert models traffic system components in DEVS and encodes them as atomic and coupled DEVS. This *simulation logic library* defines the behavior of each component. A traffic DSM, instance of the DSL, is automatically generated into a specific DEVS model (a coupled DEVS and coupling of its sub-models) and the experiment frame. This corresponds to the first step in Figure 1.

The second step is to simulate the generated DEVS model. The traces of execution of the simulation are stored in a log.

The third step happens offline, meaning after the simulation has terminated. In this step the trace log is filtered and transformed into an animation model. It dictates the sequence of statements to manipulate the DSM.

Finally, the instructions are sent to the AToMPM server in order to visually animate the DSM. In the following, we outline each component of the architecture in Figure 1 and their relationships.

DSM is a domain-specific model that conforms to an explicitly modeled domain-specific language. In our example, this is a model of the traffic DSL. It involves concepts proper to traffic network modeling, independent from DEVS. The DSM is modeled in AToMPM.

DS-DEVS PIM is a domain-specific DEVS platform-independent model. It is a DEVS model specific to the domain, but not specific to any DEVS implementation.

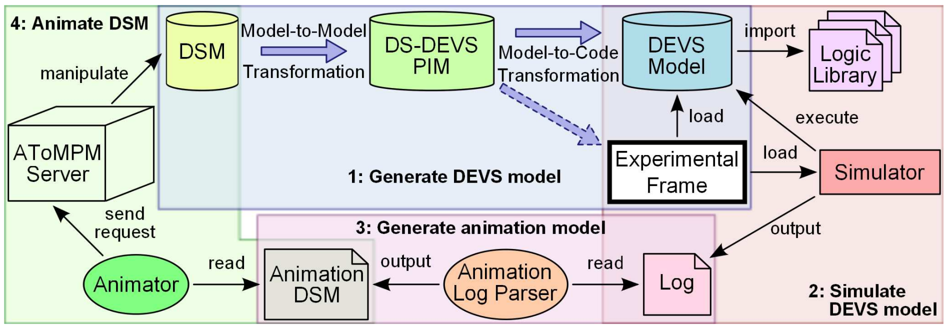


Figure 1. The overall architecture.

It is an abstraction of the coupled models in the logic library. In our example, this is a refinement of the traffic DSM mapped onto DEVS-specific constructs. A rule-based model transformation in AToMPM defines the mapping from each entity in the traffic DSL onto their representation in DEVS. This transformation is essential for the animation as it defines the correspondence between AToMPM objects and DEVS objects.

DEVS Model is a particular instance of the atomic and coupled DEVS entities defined in the logic library. In our example, this is the traffic network model in PythonDEVS. It consists of instances of coupled DEVS the traffic library defines. The code generation, also implemented in AToMPM, produces the configuration of the DEVS model, not its behavior. It also specifies the initial state of atomic DEVS models.

Logic Library encodes, for a target language implementation of DEVS, the functionality of the system to simulate as atomic and coupled DEVS entities. In our example, the traffic library defines all atomic DEVS models, their external/internal transitions and their output/time advance functions in Python.

Experimental Frame is partially generated from the DS-DEVS PIM. It defines the initial parameters of the DEVS model as well as the setup for the logging mechanism. In our example, it specifies the termination conditions of the DEVS traffic network model, initializes the static parameters and links the log to the simulator.

Simulator is a DEVS simulator that executes the DEVS model. In our example, we used the PythonDEVS classical simulator.

Log collects all log events the simulator outputs. A log event is generic to any DEVS simulation.

Animation Log Parser filters the log events that are only needed for animation. It produces the corresponding animation model. In our current implementation, the animation log parser is not real-time: the log parser starts after the simulation is complete. The benefit is that one can replay the animation without having to run the simulation every time.

Animation DSM is a domain-specific model that specifies what changes need to happen in a DSM to animate it. It is generic for animating any DSM for AToMPM. Currently, the animation DSM is a sequence of statements that indicate the steps of the animation. A statement can create a model element, update some of its attribute value(s), delete the element, or modify its geometric location.

Animator takes as input an animation DSM and transforms it into a series of HTTP requests to the AToMPM server. Ideally, this would be a model transformation generated from the animation DSM model through higher-order transformation. However, we have currently implemented as a plugin of AToMPM.

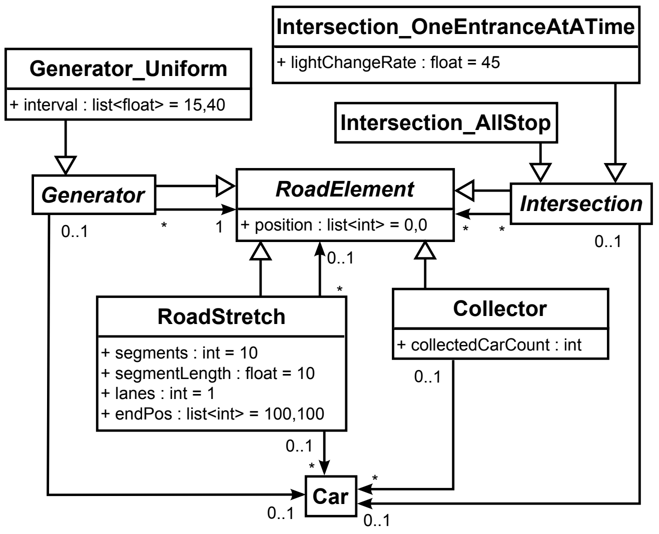
AToMPM Server handles any request received to manipulate a model and propagates the change to the views of all registered clients.

3. TRAFFIC SIMULATION MODEL

In the following we outline the traffic DSM and its implementation in DEVS.

3.1. Traffic DSL to DEVS

Figure 2 illustrates the traffic DSL which focuses solely on traffic network concepts, with no notion of DEVS. The model-to-model transformation mentioned previously refines an instance of this DSL into a DS-DEVS PIM defined by the meta-model in Figure 3). This transformation simplifies the modeler's task of creating a valid and optimized model (the most notable step being the expansion of RoadStretches into two sub-types). The model-to-code transformation is a simple one-to-one mapping of each DS-DEVS PIM entity to a DEVS entity defined in the logic library. The framework currently supports DEVS simulators written in Python (which we chose for its excellence as a prototyping language) and will eventually support more.



Intersection_OneEntranceAtATime
+ lightChangeRate : float = 45
Generator_Uniform
+ interval : list<float> = 15,40
Intersection_AllStop
Generator
RoadElement
+ position : list<int> = 0,0
Intersection
RoadStretch
+ segments : int = 10
+ segmentLength : float = 10
+ lanes : int = 1
+ endPos : list<int> = 100,100
Collector
+ collectedCarCount : int
Car

Figure 2. The Traffic DSL.

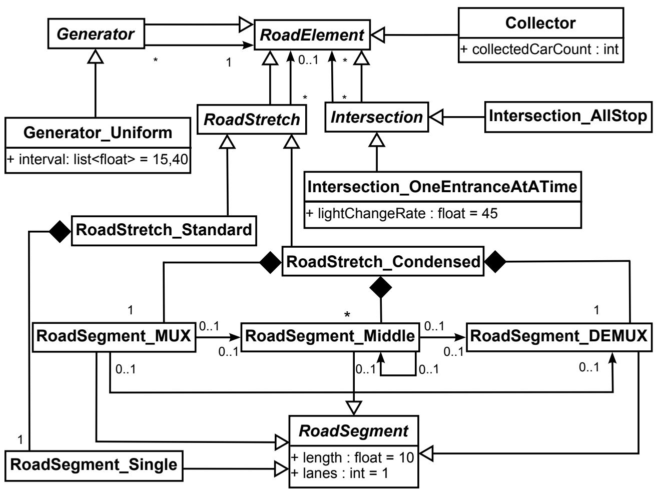


Figure 3. The DS-DEVS platform-independent meta-model.

3.2. DEVS Traffic Simulation Model

Before proceeding, it is necessary to describe the individual components of the Traffic DEVS meta-model.

Road Element encodes an atomic or coupled DEVS entity that can send or receive cars. A traffic network model consists of a partially sequence of road elements.

Cars are the fundamental component of the traffic DEVS meta-model. They are implemented as messages passed between road elements. Cars also encode the typical driver's approximation attributes, such as: current velocity, preferred velocity, minimum and maximum ac/deceleration, and front and rear spacing.

Query is a message sent from a road element to the "next" road element, encoding the driver of a specific car wishing to move to the target road element. It request information regarding the presence of other cars. Once a query has been received, its querier (the car) is registered such that future changes are also propagated to the requesting road element.

Acknowledgment is a message sent one or more times in response to a query. The response specifies the time until a slot opens (`timeUntilOpen`) for the querying car to move ahead.

Road Segment is an atomic DEVS with one or more lanes. Each lane may contain cars; however, we consider the presence of two or more cars in a lane as being in a "crashed" state. Conceptually, each road segment will be described as owning six ports per lane: query, acknowledgment, and car in/outports, each set connected to an arbitrary destination. In practice, however, we have unified these ports where possible to improve the simulation performance.

Road Stretch is a coupled road element consisting of one or more road segments in succession. There are two variants of road stretch: multiple port and condensed port.

Intersection is a road element capable of directing the cars of many incoming road elements to many outgoing road elements. The manner in which it directs traffic is defined by its sub-classes.

3.2.1. High-Level Design

In contrast with agent-based traffic system models [7], we chose to model static elements (such as roads, intersections, and buildings[1]) as atomic and coupled DEVS and cars as messages passed between them. This is an effective arrangement, since the vast majority of drivers' dynamic concerns are within their immediate vicinity; this allows them to be modeled with simple query and acknowledgment messages passed between neighboring DEVS.

While drivers do have physically distant concerns (*e.g.,* pathing towards distant locations), the majority of these can be modeled with elements that are static with respect to the simulation; therefore, they can be modeled independently of DEVS, though still in conjunction with it. This is possible since the information provided by static elements of the meta-model do not change during the simulation, and thus do not depend on it. For example, when a driver's path becomes obstructed, he could reroute by performing an A* search on the traffic system, *without* violating the principles of DEVS, since a traffic system's structure remains unchanged throughout its simulation. This example is a planned feature of the simulation.

There are several drawbacks to this approach, however. Namely, cars should not be aware of dynamic elements further than a single DEVS away. Hence, road segments were

[1] We have currently modeled buildings with generators or collectors.

designed encompassing multiple lanes, allowing cars to be completely aware of others in the same segment, no matter how many lanes away. This decision was motivated by a problem in the initial design of our meta-model, as road segments consisted of only one lane. Because of this, cars were unaware of other cars more than a lane away, which caused them to blindly merge into the same lane at once. When left unchecked, the system was plagued with unrealistic crashes; however, when a complex system of queries was implemented to remedy this, the system became terribly slow. Our solution, however, has proven to be highly scalable as the number of lanes increases, and we remain confident that future, related difficulties can be overcome by analogous solutions.

3.2.2. Road Element Behavior

When a car first arrives at a road element, its next destination must be determined. By "destination," we mean the road element out of the set of possible outgoing road elements that the car will be sent to. To do this, it performs the three operations upon arrival: (1) decide destination from set of outgoing road elements, (2) query the destination road element, and (3) maintain current velocity. Then, the car will either receive an acknowledgment from its destination after some elapsed time or leave the segment before ever being acknowledged (the consequence of poor visibility, distraction, etc). The former would constitute normal behavior. The latter, however, may result in a crash, since the car could not react to any information regarding its destination.

The latter behavior is of particular interest, since it allows us to model a number of factors with ease: cars may have intermittent periods of distraction, occurring with uniform randomness (as with texting while driving) or beginning at predefined locations (as with picking up children after school), and turns can be modeled with poor visibility due to foliage. Where possible, factors are parameterized on an individual driver basis, such that increased distraction can be allotted to teenagers, or slower reaction time allotted to the elderly.

If and when a car is acknowledged, the road element will determine if the car can arrive at its destination safely. Then, if it can arrive safely, the car's velocity is adjusted. Otherwise, the road element chooses another destination. If at this point, all destinations have been exhausted, then the car is informed to stop. Otherwise, the new destination is queried, and the car is informed to slow down.

Our definition of "safely" is somewhat more involved than a simple crash check, however. Specifically, each car will attempt to respect its preferred front and rear spacing, both of which are parameterized; higher spacings emulate more cautious driving, whereas lower spacings emulate recklessness. The front spacing is applied when gauging the departure time of the car inhabiting the current destination, whereas the rear spacing is applied when observing the earliest departure time of other cars headed to that destination. If the driver can adjust its velocity such that both conditions will be satisfied, the car adjusts accordingly and prepares for its destination; otherwise, a reroute occurs.

If the road element has exhausted every possible destination, it will attempt to stop the car before it leaves. If it succeeds, the car will remain stationary, and the segment will send queries to potential destinations at increasing intervals until one becomes available (analogous to the driver whose attentiveness wanes while waiting for traffic to budge.) Otherwise, the car will enter its last destination unintentionally and likely crash as a consequence.

3.2.3. Intersection Behavior

The behavior of an intersection differs from that of the standard road element. Specifically, an intersection may respond to queries with acknowledgments of infinite `timeUntilOpen`, such that recipient cars will come to a halt (as in the case of a stop sign). Furthermore, for the sake of reducing needless complexity, intersections transmit contained cars instantaneously, such that at most one car is ever contained at a time. For this reason, it is important that the *select* function is designed carefully, such that an intersection always sends any contained car before receiving another. This models both stop sign and semaphore intersections.

3.2.4. Queries as Registration

When a road element receives a car, it must inform any prior lanes that have recently sent queries of the change. For instance, when a car accelerates to beat another car to its destination (as in the previous description), the other car must be notified of this change, lest it were to crash unrealistically. Each road element accomplishes this by maintaining a list of cars and their destinations, each index of the list being associated with a previous lane. When the road segment receives a query, the querying car is"registered" into the array by the index of its current lane. Then, when a car enters a lane, all cars in its road segment's list whose destinations are the current lane are notified. A car is only removed from the list once it has arrived in the road segment; however, there are a number of other cases in which a car could be removed. Since a car may never enter the road segment it queries (*e.g.,* when a crash is imminent), a road segment may continue to notify previous lanes of car arrivals, despite the previous lanes being vacant. Additional methods of removing unnecessary cars from the list are an open area of study.

3.2.5. Combining Road Segment Ports

As previously mentioned, a road segment's ports are frequently combined as a performance measure. This is the case when the modeler creates a condensed port road

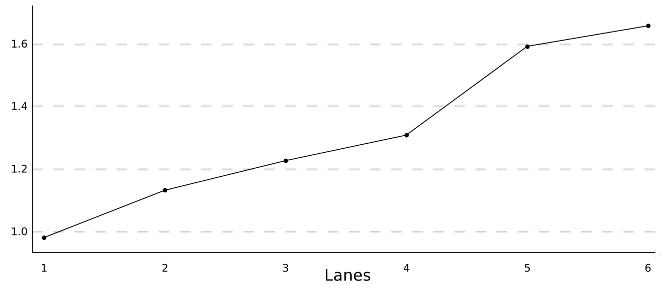


Figure 4. Condensed vs Multiple Port Performance.

stretch, whose segments may be one of three variants: MUX, Middle, and DEMUX. The first road segment is a MUX, the last is a DEMUX, and all other segments are Middles. As in digital circuitry, a MUX combines multiple queries/acknowledgment streams into one, whereas a DEMUX splits them back into many. This is accomplished by using variants of query and acknowledgment, each with two additional fields, `laneFrom` and `laneTo`. For road stretches with more than one lane, the performance gains are significant as redundant external transitions are combined, reducing the overall workload on the DEVS simulator as demonstrated in Figure 4.

3.2.6. Road Element Select Functions

In *select* functions where the imminent set contains road elements, it is desirable that a "foremost" road element be selected. That is, an imminent road element whose transition is guaranteed *not* to result in the *external transition* of another imminent element. This is to prevent erroneous corner cases, such as when a car is scheduled to advance the instant the car at its destination is scheduled to leave, and is selected *before* the destination car has a chance to leave.

In a road stretch, selecting the foremost element is trivial, since its elements are simply items in a list, the foremost element being the closest to the end. In a coupled DEVS containing a complicated network, however, it is necessary to perform a topological sort on the nested DEVS to efficiently discover an ordering of foremost elements (if there is one). If the graph of nested DEVS is acyclic, then a total ordering exists, and a sort need only be performed once before the simulation begins. If the graph contains a cycle, however, then a modified topological sort must be performed within the *select* function such that cycles containing nodes not in the imminent set are ignored.

4. LOGGING AND ANIMATION

We now describe how this DEVS model will generate an animation for AToMPM to render.

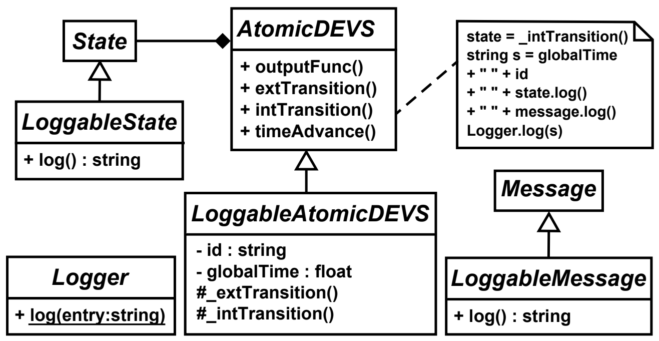


Figure 5. Extension of DEVS to allow logging.

4.1. Generating the Log

Each log entry represents either an *external* or *internal transition* of an atomic DEVS model. An external entry consists of a time, id, state, and message, whereas an internal entry consists of only a time, id, and state. The time is simply the simulation time at which the event occurred. The id is a unique identifier given to the DEVS when it is generated, so that information pertaining to it may be communicated back to AToMPM. The state and message are Traffic DEVS element specific (*e.g.*, in the case of a traffic light, the state is a string representing the current color of the light).

Logging functionality is added to the traffic DEVS meta-model instead of the simulator in order to keep the freedom of selecting any DEVS simulator (classic, parallel, real-time, etc.). We extend the meta-model of DEVS via inheritance (illustrated by Figure 5), such that each atomic and coupled DEVS sub-class inherits from a *loggable* super-class. The *external* and *internal transitions* are overridden to invoke their implemented counterparts and write the log entry.

4.2. Animation

After producing the event log, the entries of the log are filtered such that only those relevant to the animation are obtained. They are then transformed into a simplified animation DSM. This is interpreted by the animator, which is in constant communication with AToMPM via HTTP requests. Specifically, AToMPM is able to direct the actions of the animator such that the modeler may play and replay the animation through AToMPM's GUI, and the animator is able to communicate transitions of the animation in accordance with this.

Each statement of the animation DSM takes the form `<time> <ID> <newState>`, where `newState` is a string corresponding to an animation state of an entity in AToMPM. For example, a drawbridge would have two states, 'raised and lowered, each with differing animations. A statement might read `637.00 45 raised`. Figure 6 depicts a snapshot of a traffic DSM animated in AToMPM.

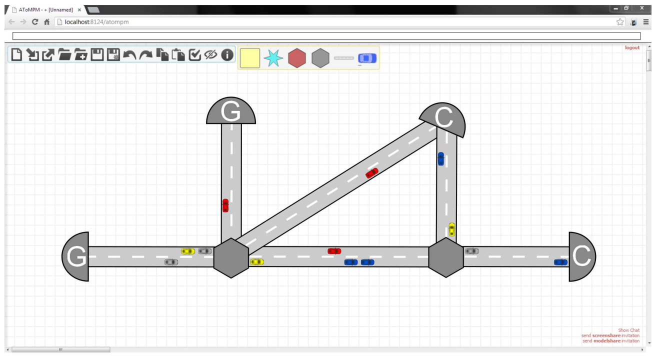


Figure 6. A snapshot of the animated traffic DSM.

5. RELATED WORK

Recently, DEVSML 2.0 [4] introduced the idea of first letting a domain expert develop a DSL and then define a model transformation to DEVS, which requires DEVS expertise. The DSL is transformed into DEVSML, a platform-independent representation of DEVS which is then compiled and executed by a particular DEVS simulator. Ideally in our framework, the logic library can be modeled in DEVSML, instead of coded in PythonDEVS. DEVSML can also be incorporated in our framework as an intermediate between the domain-specific DEVS PIM and the generated DEVS model. The advantage gained would be to not rely on any implementation of DEVS (Python, Java, C++) and therefore promote cross-platform re-use. However, although DEVSML is a modeled language, the modeler is still required to code the functions (*internal*, *external*, *select*) in Java.

DEVSView++ [8] offers a logging mechanism for visualizing the simulation of cellular DEVS models. The logs are triggered after either an external transition or an output function is executed. As soon as a log event is created it is sent to a dedicated parser which extracts the required information to visualize the model. The logging system described in this paper differs in the one of DEVSView++ in various ways. DEVSView++ logger is not well suited for classical DEVS models since it does not consider states resulting from an internal transition (because DEVSView++ was designed for cellular DEVS instead). Furthermore, in AToMPM, animation of models must be triggered by HTTP requests. Therefore parsing each log event as they are created does not guarantee that the simulation time is respected due to unpredictable network delays.

6. CONCLUSION

In this paper, we have outlined a framework for animating the execution of traffic simulation DEVS models. This enables domain experts to visualize and ultimately unit test their models.

The foreseen bottleneck of the presented architecture is the scalability issue to large complex traffic models. However, since unit testing typically involves a small subset of the model, this framework gives correct feedback in a reasonable amount of time for the simulation tester. In order to build the complete DEVS simulation testing framework, we will investigate how to define test oracles, how to specify failure and sanity testing, and report on the usefulness of this testing approach.

We would also like to further improve the automation of the framework. Recall that the animation log parser requires the mapping from DEVS elements back to the DSM. Our current implementation of the prototype keeps a dictionary mapping AToMPM object identifiers to atomic and coupled DEVS model identifiers. In the future, we plan to be able to automatically extract the reverse mapping from the model-to-model transformation. This can be done, for example, by a higher-order transformation that takes as input the DSM to DS-DEVS PIM transformation and feeds the identifier mapping directly to the animation log parser.

REFERENCES

[1] S. Algers et al., “Review of Micro-Simulation Models,” University of Leeds, Institute for Transportation Studies, SMARTEST Deliverable 3, 1997.

[2] J. Gray, J.-P. Tolvanen, A. Kelly, Steven Gokhale, S. Neema, and J. Sprinkle, *Domain-Specific Modeling*. CRC Press, 2007, ch. chapter 7, pp. 1–20.

[3] B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[4] S. Mittal and S. A. Douglass, “DEVSML 2.0: the language and the stack,” in *Symposium on Theory of Modeling and Simulation*. San Diego: Society for Computer Simulation International, April 2012, pp. 1–12.

[5] J.-S. Bolduc and H. Vangheluwe, “The Modelling and Simulation Package pythonDEVS for Classical Hierarchical DEVS, McGill University, TR-2001–01,” 2001.

[6] R. Mannadiar, “A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling,” Ph.D. Thesis, McGill University, June 2012.

[7] P. A. M. Ehlert and L. J. M. Rothkrantz, “Microscopic traffic simulation with reactive driving agents,” in *IEEE Intelligent Transportation Systems Conference*, Oakland, August 2001, pp. 861–866.

[8] W. Venhola and G. Wainer, “DEVSView: A Tool for Visualizing CD++ Simulation Models,” in *DEVS Integrative M&S Symposium*, Huntsville, 2006.