Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

THE FUNDAMENTALS OF DOMAIN-SPECIFIC SIMULATION LANGUAGE ENGINEERING

Simon Van Mierlo
Hans Vangheluwe

Department of Mathematics and Computer Science
University of Antwerp - Flanders Make vzw
Middelheimlaan 1
Antwerp, 2020, BELGIUM

Joachim Denil

Department of Electronics - ICT
University of Antwerp - Flanders Make vzw
Groenenborgerlaan 171
Antwerp, 2020, BELGIUM

ABSTRACT

Simulationists use a plethora of modelling languages. General-purpose languages such as C, extended with simulation constructs, give the user access to abstractions for general-purpose computation and modularization. The learning curve for experts in domains that are far from programming, however, is steep. Languages such as Modelica and DEVS allow for a more intuitive definition of models, often through visual notations and with libraries of reusable components for various domains. The semantics of these languages is fixed. While libraries can be created, the language's syntax and semantics cannot be adapted to suit the needs of a particular domain. This tutorial provides an introduction to modelling language engineering, which allows one to explicitly model all aspects –in particular, syntax and semantics– of a (domain-specific) modelling and simulation language and to subsequently synthesize appropriate tooling. We demonstrate the discussed techniques by means of a simple railway network language using AToMPM, a (meta)modelling tool.

1 INTRODUCTION

This tutorial introduces the fundamentals of language engineering for creating domain-specific simulation languages.

1.1 Why Domain-Specific Language for Simulation?

The complexity of the systems we develop is continuously increasing. To overcome this complexity, the designers of the system can employ simulation techniques, building virtual prototypes of the system that allow for the quick evaluation of candidates. The structure and behaviour of these prototypes are expressed in *models* of the system: abstractions that specify a particular view on, or a part of the system. Such models are implemented in a *modelling language*, often a general-purpose programming language such as C or Java. This may lead to certain issues. The domain expert has deep knowledge of the problem domain, but only a limited understanding of computer programs. This can result in communication problems, such as the programmer making false assumptions about the domain, or the domain expert to gloss over details when explaining the problem to the programmer. Furthermore, the domain experts will finally receive a software component that they don't fully understand, making it difficult for them to validate and modify if necessary. There is a conceptual gap in effect between the two domains, hindering productivity. Alternatively, tools such as FlexSim (https://www.flexsim.com/), Simio (https://www.simio.com/), or Arena (https://www.arenasimulation.com/) offer a discrete-event abstraction with the possibility for building (domain-specific) libraries. While these tools already bridge the gap between concept and implementation partially, their semantics is fixed (discrete-event) and cannot be extended or changed.

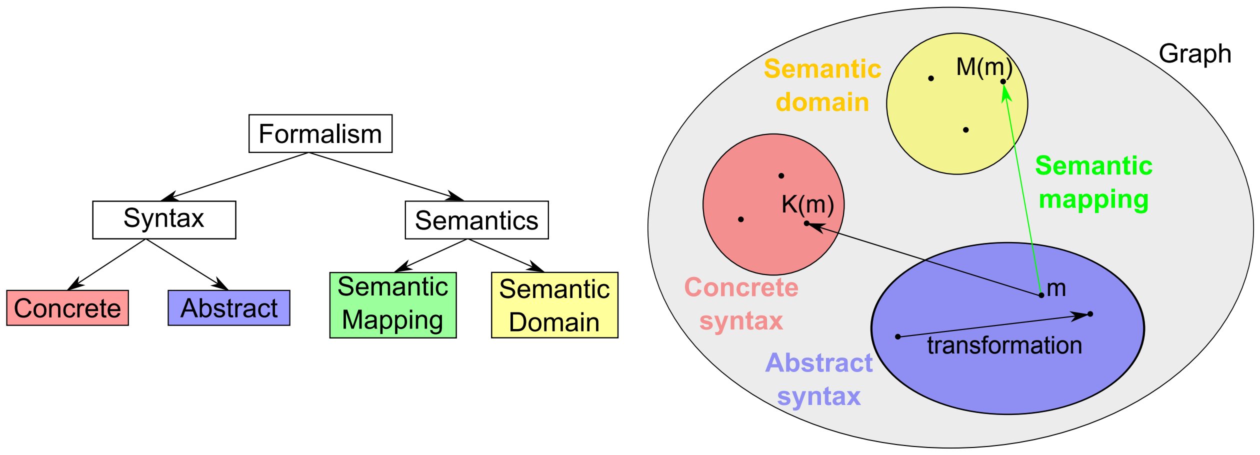


Figure 1: Terminology.

Model-Driven Engineering (MDE) (Vangheluwe 2008) tries to bridge this gap, by shifting the level of specification from computing concepts (the "how") to conceptual models or abstractions in the problem domain (the "what"). Domain-Specific Modelling (DSM) (Kelly and Tolvanen 2008) in particular makes it possible to specify these models in a Domain-Specific Modelling Language (DSML), using concepts and notations of a specific domain. The goal is to enable domain experts to develop, understand, and verify models more easily, without having to use concepts outside of their own domain. It allows the use of a custom visual syntax, which is closer to the problem domain, and therefore more intuitive. Models created in such DSMLs are used, among others, for simulation, (formal) analysis, documentation, and code synthesis for different platforms. There is, however, still a need for a language engineer to create the DSML, which includes defining its syntax, and providing the mapping between the problem domain and the solution domain.

This tutorial introduces language engineering for domain-specific simulation languages. A DSML is fully defined (Kleppe 2007) by:

1. Its *abstract syntax*, defining the DSML constructs and their allowed combinations. This information is typically captured in a metamodel.
2. Its *concrete syntax*, specifying the visual representation of the different constructs. This visual representation can either be graphical (using icons), or textual.
3. Its *semantics*, defining the meaning of models created in the domain (Harel and Rumpe 2004). This encompasses both the *semantic domain* (*what* is the meaning of models in the DSML), and the *semantic mapping* (*how* to give meaning to the models in the DSML).

For example, $1+2$ and $(+\ 1\ 2)$ can both be seen as textual concrete syntax (*i.e.*, a visualization) for the abstract syntax concept "addition of 1 and 2" (*i.e.*, what construct it is). The semantic domain of this operation is the set of natural numbers (*i.e.*, what it evaluates to), with the semantic mapping being the execution of the operation (*i.e.*, how it is evaluated). Therefore, the semantics, or "meaning", of $1+2$ is 3.

This definition of terminology can be seen in Fig. 1. Each aspect of a formalism is modelled explicitly, as well as relations between different formalisms.

1.2 Multi-Paradigm Modelling

Multi-Paradigm Modelling (MPM) (Mosterman and Vangheluwe 2004) aims to support the modelling of complex systems by combining three different directions of research:

- *Meta-Modelling*, which is the process of modelling formalisms.

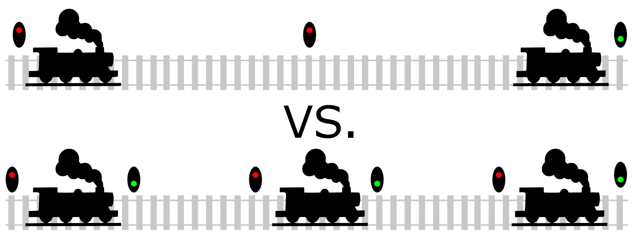


Figure 2: The optimal track length problem illustrated.

- *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
- *Multi-Formalism Modelling*, concerned with the coupling of and transformation between models described in different formalisms.
- *Process Modelling*, allowing to describe system development workflows.

This allows developers to model every aspect of interest explicitly, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, with processes explicitly modelled.

In many cases, an appropriate formalism (with respect to some properties such as the expressiveness, tool support, and other pragmatics such as the knowledge the development team has of the language) is readily available. Examples include Matlab/Simulink for specifying the dynamics of system using a block-diagram abstraction, Modelica for modelling the physics of systems in an acausal way, and DEVS for discrete-event systems such as queuing systems. In other cases, such a language is not readily available and one has to be created to serve for a specific domain, or to solve a specific problem. Within MPM, meta-modelling allows a language engineer to efficiently build such languages and generate domain-specific modelling environments. Next to the specification of the syntax of the language, a meaning is given to the models of the language by specifying the semantics of the language.

1.3 Example: Railway Simulation

The tutorial uses an example from the railway domain as an application of the presented techniques. When designing railway networks, domain experts create the network by connecting railway building blocks, such as:

- *Stations*, where people can get on an off the trains.
- *Track Segments*, connecting other track segments to each other, as well as to stations.
- *Straights*, a special type of track that has exactly one incoming segment and one outgoing segment.
- *Turnouts*, a special type of track that splits into two other tracks.
- *Junctions*, a special type of track that merges two other tracks.
- *Lights*, controlling whether or not a train can enter the next track segment.

Such models might be subject to evaluation trough virtual experiment (simulation) to see whether some performance metric is optimal. For example, one could ask the question what the optimal track length is, given a specific railway network. The track length influences the total cost of the system: very long tracks mean less infrastructure has to be built (since there are less lights to put next to the tracks), but it also means that trains might have to wait very long before they are admitted to enter the next track, potentially causing delays.

To answer such questions, the models are typically encoded in an appropriate modelling language, where such answers can be easily given. The optimal track length problem is illustrated in Figure 2. Domain experts might be more comfortable with such visual models, instead of a simulation language that is specifying *how* to run the simulation. This can improve understandability, both for expressing the problem correctly, and for interpreting the results of simulation.

2 WHAT IS A DOMAIN-SPECIFIC MODELLING LANGUAGE?

A domain-specific modelling language is a modelling language, tailored to a specific domain. The concept of domain-specificity is relative: a language such as HTML/CSS can be viewed as domain-specific, since it is tailored to the specification of the structure and layout of webpages. However, these languages are very general, and can specify solutions in many domains. The explanations that follow can be applied to modelling languages in general, regardless of whether they are viewed as domain-specific or not.

In the following subsections, we explain the building blocks of domain-specific languages.

2.1 Syntax

A syntax defines whether elements are valid in a specified language or not. It does not, however, concern itself with what the constructs mean. With syntax only, it would be possible to specify whether a construct is valid, but it might have invalid semantics. A simple, textual example is the expression $\frac{1}{0}$. It is perfectly valid to write this, as it follows all structural rules: a fraction symbol separates two recursively parsed expressions. However, its semantics is undefined, since it is a division by zero.

2.1.1 Abstract Syntax

The abstract syntax of a language specifies its constructs and their allowed combinations, and can be compared to grammars specifying parsing rules. Such definitions are captured in a metamodel, which itself is again a model of the metametamodel (Kühne 2006). Most commonly, the metametamodel is similar to UML Class Diagrams, as is also the case in our case study. The metametamodel used in the examples makes it possible to define classes, associations between classes (with incoming and outgoing multiplicities), and attributes.

While the abstract syntax reasons about the allowable constructs, it does not state anything about how they are presented to the user. In this way, it is distinct from textual grammars, as they already offer the keywords to use (Kleppe 2007). It merely states the concepts that are usable in the domain.

2.1.2 Concrete Syntax

The concrete syntax of a model specifies how elements from the abstract syntax are visually represented. The relation between abstract and concrete syntax elements is also modelled: each representable abstract syntax concept has exactly one concrete syntax construct, and vice versa. As such, the mapping between abstract and concrete syntax needs to be a bijective function. This does not, however, limit the number of distinct concrete syntax definitions, as long as each combination of concrete and abstract syntax has a bijective mapping. The definition of the concrete syntax is a determining factor in the usability of the DSML (Barišić, Amaral, Goulão, and Barroca 2011).

Multiple types of concrete syntaxes exist, though the main distinction is between *textual* and *graphical* languages. Both have their advantages and disadvantages: textual languages are more similar to programming languages, making it easier for programmers to start using the DSML. On the other hand, visual languages can represent the problem domain better, due to the use of standardized symbols, despite them being generally less efficient (Petre 1995). An overview of different types of graphical languages is given in (Costagliola, Deufemia, and Polese 2004). Different tools have different options for concrete syntax, depending on the expected target audience of the tool. For example, standard parsers always use a textual language, as their target audience consists of computer programmers who specify a system in (textual) code.

While the possibilities with textual languages are rather restricted, graphical languages have an almost infinite number of possibilities. In (Moody 2009), several “rules” are identified for handling this large number of possibilities. As indicated beforehand, a single model can have different concrete syntax representations, so it is possible for one to be textual, and another to be graphical.

2.2 Semantics

For a domain-specific language to be called a formalism (Giese, Levendovszky, and Vangheluwe 2007), it requires a semantic definition. Since the syntax only defines what a valid model looks like, we need to give a meaning to the models. Even though models might be syntactically valid, their meaning might be useless or even invalid.

It is possible for humans to come up with intuitive semantics for the visual notations used (*e.g.*, an arrow between two states means that the state changes from the source to the destination if a certain condition is satisfied). There is, however, a need to make the semantics explicit for two main reasons:

1. Computers cannot use intuition, and therefore there needs to be some operation defined to convey the meaning to the machine level.
2. Intuition might only take us that far, and can cause some subtle differences in border cases. Having semantics explicitly defined makes different interpretations impossible, as there will always be a "reference implementation".

Semantics consists of two parts: the domain it maps to, and the mapping itself. While many categories of semantic mapping exist, as presented in (Zhang and Xu 2004), we only focus on the two main categories relevant to our case study:

1. *Translational semantics*, where the semantic mapping translates the model from one formalism to another, while maintaining an equivalent model with respect to the properties under study. The target formalism has semantics (again, either translational or operational), meaning that the semantics is "transferred" to the original model.
2. *Operational semantics*, where the semantic mapping effectively executes, or simulates, the model being mapped. Operational semantics can be implemented with an external simulator, or through model transformations that simulate the model by modifying its state. The advantage of in-place model transformations is that semantics are also defined completely in the problem domain, making it suitable for use by domain experts. For our case study, this means implementing a simulator using model transformations.

Both the semantic domain and the possible semantic mappings (operational and translational) will be covered in the next subsections.

2.2.1 Semantic Domain

The semantic domain is the target of the semantic mapping. As such, the semantic mapping will map every valid language instance to a (not necessarily unique) instance of the semantic domain. Many semantic domains exist, as basically every language with semantics of its own can act as a semantic domain. The choice of semantic domain depends on which properties need to be conserved. For example, DEVS (Zeigler, Praehofer, and Kim 2000) can be used for simulation, Petri nets (Murata 1989) for verification, Statecharts (Harel 1987) for code synthesis, and Causal Block Diagrams (Cellier 1991) for continuous systems using differential equations. A single model might even have different semantic domains, each targeted at a specific goal.

For the example simulation language developed during the tutorial, we will use DEVS as the semantic domain; we want to run several simulations to analyze the optimal track length, which requires to look at the overall cost of the system using the throughput of the track by regarding it as a queueing system. DEVS is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction.

Atomic DEVS models are the behavioural atomic blocks of a DEVS model. Their structure is described by:

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

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

Simulating an atomic DEVS model results in a (discrete-event) trace of output events. This allows a modeller to check the behaviour of the system by, for example, computing some performance metric.

Coupled DEVS models as the structuring concept of DEVS. Their structure is described by:

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

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

DEVS is closed under coupling, so models can be nested up to arbitrary depth.

The semantics of DEVS, a discrete-event formalism are well known (Zeigler, Praehofer, and Kim 2000) and implemented in a number of tools such as PythonPDEVS (Van Tendeloo and Vangheluwe 2015). In the tutorial, we will use DEVS as the semantic domain for our railway language to evaluate its behaviour by looking at it as a queuing network and extracting throughput information from the different scenarios.

2.2.2 Translational Semantics

With translational semantics, the source model is translated to a target model, expressed in a different formalism, which has its own semantic definition. The (partial) semantics of the source model then correspond to the semantics of the target model. As the rule uses both concepts of the problem domain and the target domain (DEVS in our case), the modeller should be familiar with both domains.

2.2.3 Operational Semantics

A formalism is operationalized by defining a simulator for that formalism. This simulator can be modelled as a model transformation that executes the model by continuously updating its state (effectively defining a "stepping" function). The next state of the model is computed from the current state, the information captured in the model (such as state transitions and conditions), and the current state of the environment. Contrary to translational semantics, the source model of operational semantics is often augmented with runtime information. This requires the creation of both a "design formalism" and a "runtime formalism". In our case study, for example, the runtime formalism is equivalent to the design formalism augmented with information on the current state and the simulated time, as well as a list of inputs from the environment.

2.2.4 Terminology

The difference between a *language* and a *formalism* is not always clear. To scope our work, we define the vocabulary that we use:

- A *language* is the set of abstract syntax models. No meaning is given to these models.
- A *concrete language* comprises both the abstract syntax and a concrete syntax mapping function. Obviously, a single language may have several concrete languages associated with it.
- A *formalism* consists of a language, a semantic domain and a semantic mapping function giving meaning to model in the language.
- A *concrete formalism* comprises a formalism together with a concrete syntax mapping function.

Whenever we use the term *domain-specific language* in this tutorial, we assume it is a *concrete formalism*; this ensures that the commonly known term *DSL* is used.

3 HOW IS A DOMAIN-SPECIFIC LANGUAGE CREATED?

Now that we explained how a DSL is built up of syntax and semantics, we show technology with which it is possible to define these building blocks. By defining a DSL, a language engineer has the intention of building a *tool* that allows users of the language to build models, check whether they are valid instances of the model, simulate them, debug them, ... To efficiently do this, metamodelling environments offer a framework within which the syntax can be described in a metamodel, and the semantics using model transformation.

This section explains how to create a new DSL, both its syntax and its semantics. We use the metamodelling environment AToMPM (Syriani, Vangheluwe, Mannadiar, Hansen, Van Mierlo, and Ergin 2013) to demonstrate these language engineering techniques. The presented techniques are general enough to apply them in other metamodelling environments with similar capabilities as AToMPM.

3.1 Generating a Syntax-Directed Modelling Environment

The idea behind a domain-specific language is generating a domain-specific editing environment. This environment is a tool, used by domain experts, that helps them to define models that are valid according to the rules defined in the language. One way of achieving this is offering a free-hand editor (such as a text editor), equipped with parsers, syntax checkers, and compilers configured for the language. These work in a similar way to integrated development environments, such as Eclipse. They offer freedom with respect to the editing of the syntax, but, as a downside, one can only confirm the model is correct with respect to the language definition after a manual "check" is performed. Another way of achieving this is through a so-called *syntax-directed modelling environment*. These environments are specifically configured to the modelling language and have a model of its syntax that is continuously checked. This allows for the environment to only allow operations on the model that lead to a syntactically valid model. This constrains the user more, but since the language is tailored to a specific domain, the domain's constraints can be

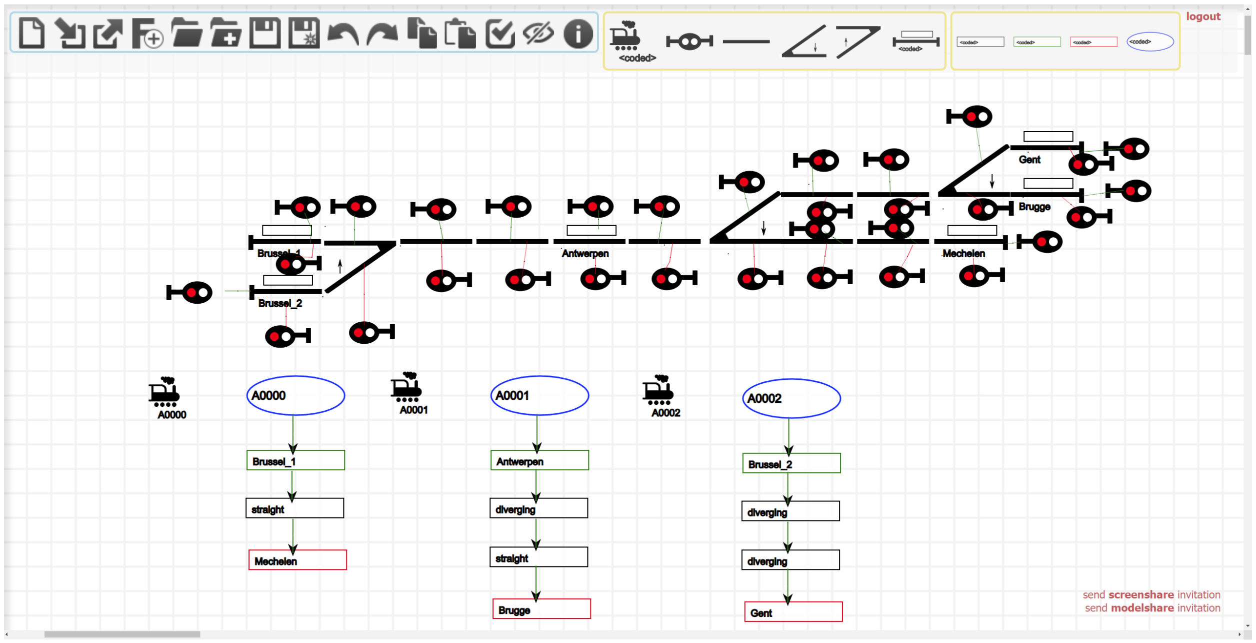


Figure 3: Example model in the domain-specific editing environment for the railway language.

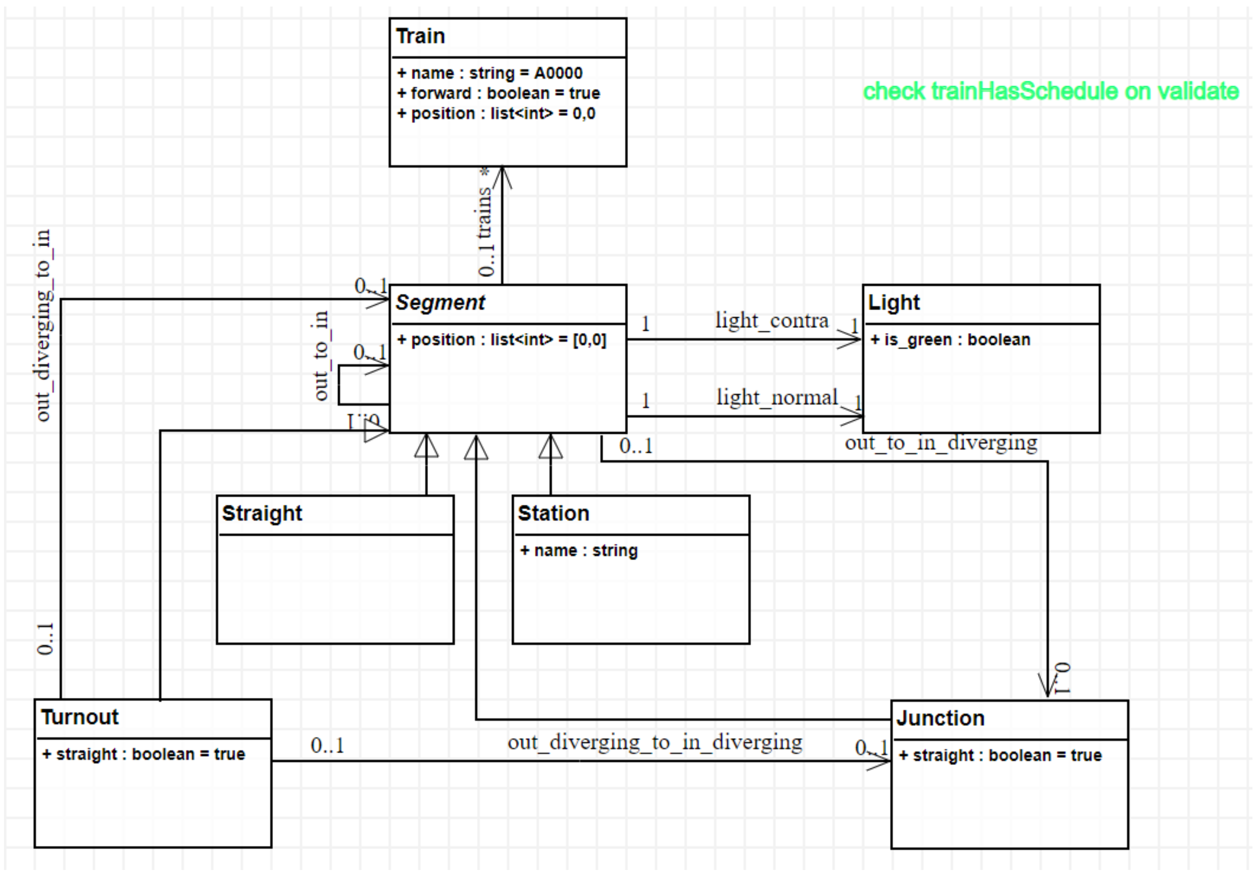
encoded in this way and improve the efficiency of users. Often, such environments are graphical; they offer the user a set of domain-specific icons and links in a toolbar that can be placed on a canvas. For domain experts, this often coincides with their intuition, especially for technical drawings in the electrical, mechanical, . . . domains.

We focus in this tutorial on generating syntax-directed graphical modelling environments. An example, for the railway language, is shown in Figure 3. It consists of a toolbar, listing the valid language element that the user of the language can instantiate. For the railway language, this includes the notions of straights, junctions, turnouts, stations, lights, and trains. On the canvas, the modeller can instantiate the graphical representation of these elements. In the example, a railway network is built consisting of a number of segments that connect stations.

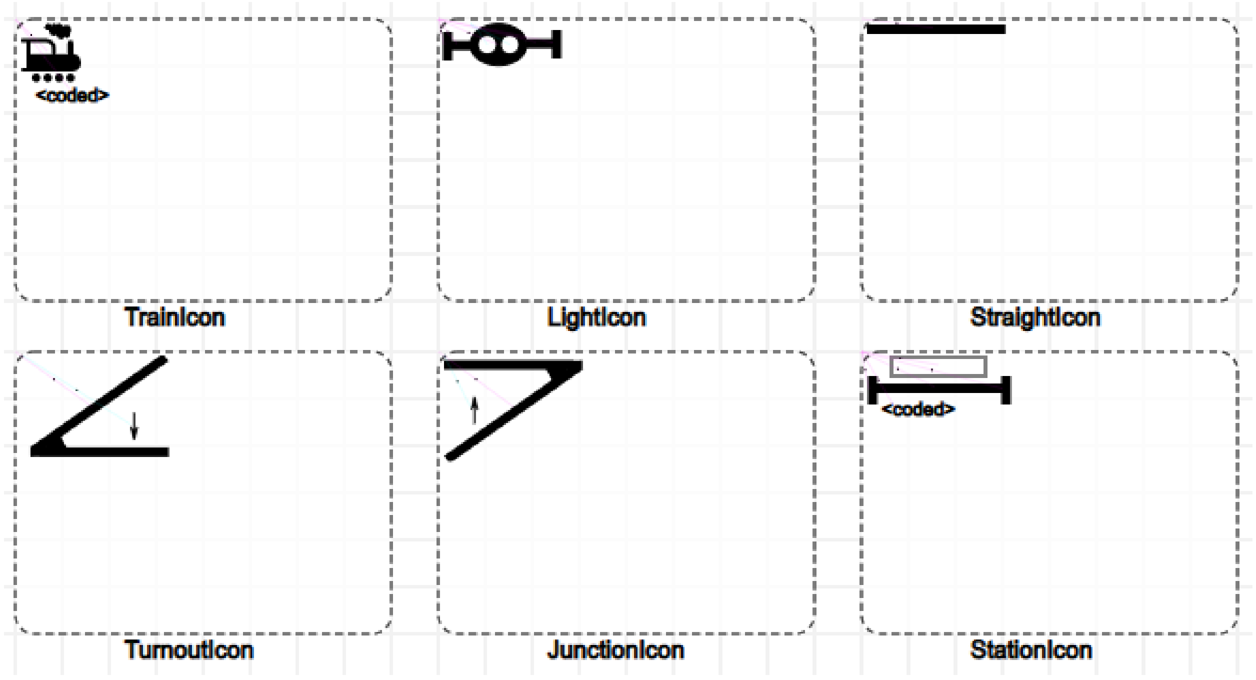
To obtain such a modelling environment, we need to specify the syntax of the language, shown in Figure 4. The metamodel (shown in 4(a)) defines the language constructs of the railway language. It also lists the allowed associations between abstract syntax elements (*e.g.*, a train cannot be connected to a light, but a segment can be connected to a train). While we don't yet give semantics to our language, we already limit the set of valid models (*i.e.*, for which semantics need to be defined). By preventing ambiguous situations in the abstract syntax, we do not need to take them into account in the semantic definition, as they represent invalid configurations.

A possible concrete syntax for the railway language, shown in Figure 4(b), assigns an icon to each langauge element. Each of the constructs shown in the concrete syntax model corresponds to the abstract syntax element with the same name. Every construct receives a visual representation that tries to be as close as possible to the mental model of the domain experts using the language. In case of standardized icons or symbols, it would be trivial to define a new concrete syntax model.

Tge metamodel together with the concrete syntax definition is used by AToMPM to generate a domain-specific syntax-directed modelling environment, which means that only valid instances can be created. For example, if the abstract syntax model specifies that a segment can only be connected to one light, trying to connect that segment to a second light will result in an error. This maximally constrains the modeller and ensures the models are (syntactically) correct by construction. Once we have defined the concrete and abstract syntax of the language, we can generate a graphical modelling environment, shown in Figure 3.



(a) The metamodel of the railway language.



(b) The icon definition of the railway language.

Figure 4: The syntax definition of the railway language: (a) abstract syntax and (b) concrete syntax.

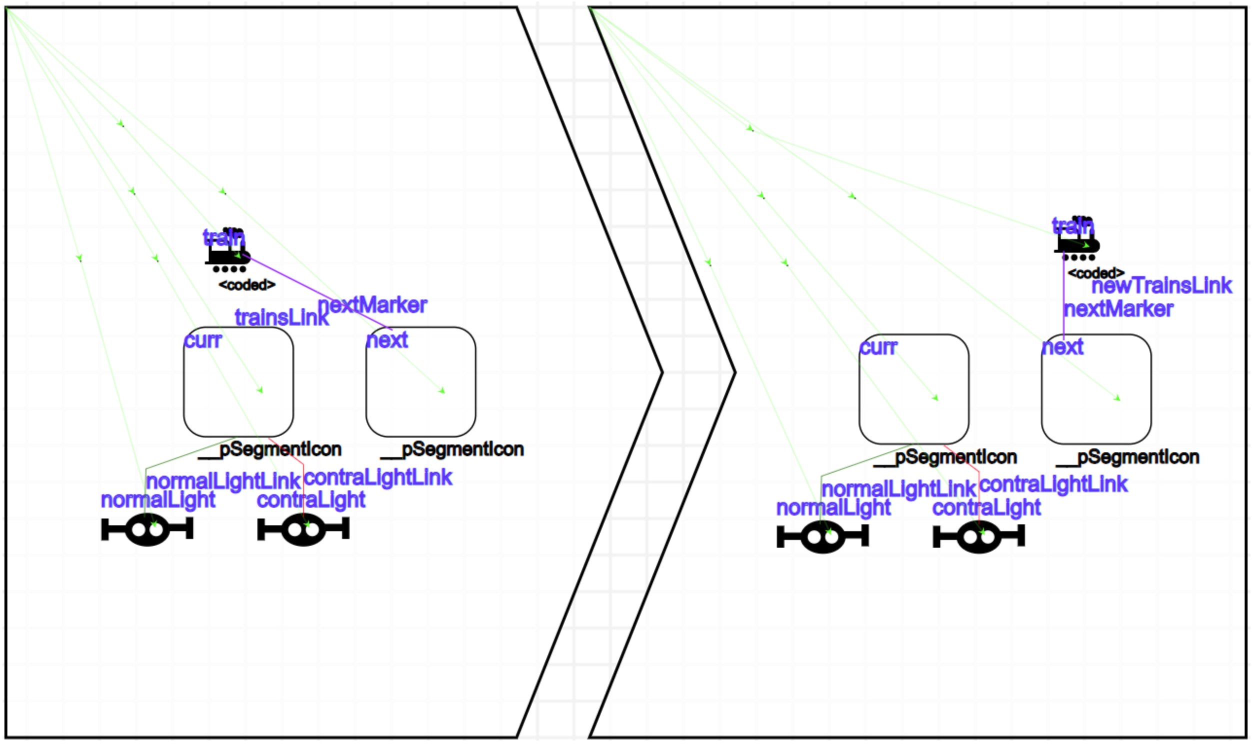


Figure 5: Example rule of the operational semantics.

3.2 Modelling the Operational Semantics

The operational semantics of a language define a simulator for that language, which defines a number of simulation *steps* that a model has to go through. It allows to obtain a simulation trace from any valid model in the language. The operational semantics are commonly expressed using model transformations, which are often called the heart and soul of Model-Driven Engineering (Sendall and Kozaczynski 2003). A model transformation is defined using a set of transformation rules, and a schedule.

A rule consists of a Left-Hand Side (LHS) pattern (transformation pre-condition), Right-Hand Side (RHS) pattern (transformation post-condition), and possible Negative Application Condition (NAC) patterns (patterns which, if found, stop rule application). The rule is applicable on a graph (the host graph), if each element in the LHS can be matched in the model, without being able to match any of the NAC patterns. When applying a rule, the elements matched by the LHS are replaced by elements of the RHS in the host graph. Elements of the LHS that don't appear in the RHS are removed, and elements of the RHS that don't appear in the LHS are created. Elements can be labelled in order to correctly link elements from the LHS and RHS.

A schedule determines the order in which transformation rules are applied. For our purpose, we use MoTiF (Syriani and Vangheluwe 2013), which defines a set of basic building blocks for transformation rule scheduling. We limit ourself to three types of rules: the *ARule* (apply a rule once), the *FRule* (apply a rule for all matches simultaneously), and the *CRule* (for nesting transformation schedules).

An example rule of the operational semantics of the railway formalism is shown in Figure 5. The left-hand side of the rule tries to find two connected tracks, where a train is currently on one of the tracks. In the right-hand side of the rule, the train is moved to the next track. This rule would appear in a schedule, that implements the complete semantics of the language. The schedule operationalizes a railway network by switching lights to red when a train enters a track, and to green again when the train leaves the track. It then moves trains along the network according to which station they want to go to, and how they want to get there (their schedule). It makes sure to check that no trains will collide by entering a track at the same

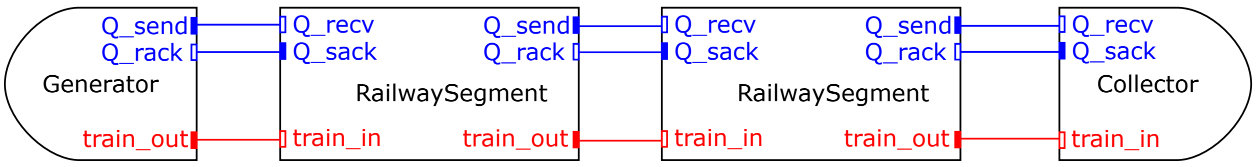


Figure 6: Result of mapping a railway model to DEVS.

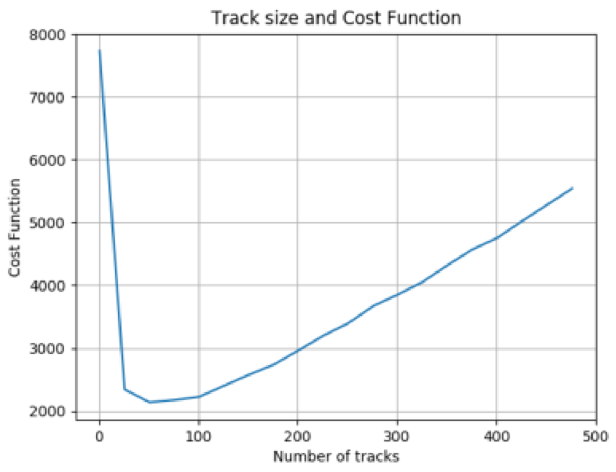


Figure 7: Analyzing the results of simulation.

time (in particular also when two trains want to enter a junction). These operational semantics “bring the system to life” and allows for users to quickly assess the behaviour of the system by visually inspecting the trace. The simulator can, moreover, compute metrics while the simulation is running, for analysis.

3.3 DEVS as a Semantic Domain for Simulation

DEVS (Zeigler, Praehofer, and Kim 2000) can be used to build queuing systems, and analyze their behaviour. For the railway system language, a particular analysis might look at the railway network as a queuing system, by looking at the tracks as connected nodes with a specific capacity; trains travel between these nodes from a start to end destination. In between, they might have to wait for red lights, causing a delay in their arrival. To analyze such properties, we could make use of the operational semantics of the language, encoding the appropriate behaviour. But sometimes, it might be easier to map to a language that already has capabilities (both syntactically and semantically) for analyzing such properties. That allows us to reuse existing infrastructure, avoiding the overhead of having to reimplement it.

As with operational semantics, we make use of model transformations for realizing this mapping. The basic idea of the mapping is to take each building block of the model (in our case, segments, trains, and lights) and map them onto building blocks of a model in the target language. The result of mapping a particular railway segment, modelled in our domain-specific language, to DEVS, is shown in Figure 6. The queuing system consists of a generator, that generates trains according to some distribution. The trains then flow through the network of segments (represented by atomic DEVS models) until they reach the collector (or end station). Along the way, a train will travel through a track by trying to accelerate (at some predefined maximum acceleration) to its preferred speed. When the train is close to the next segment, a query will be sent to the next segment, for checking whether or not the segment can be entered (*i.e.*, its light is green). If the light is red, the train will start to slow down, potentially stopping altogether. Once the light becomes green, a message will be sent, signifying that the train can continue.

While the simulation is running, multiple metrics are collected, such as the average speed of trains, the average time it takes them to traverse the complete railway network, the average time spent in one segment, etc. Such metrics can be used to calculate a cost metric. In our example, cost is calculated by looking at the average time it takes trains to pass through the complete railway network, as well as the track length. These two properties are in conflict: as tracks become longer, the cost goes down, but the travel time will go up as trains will frequently be waiting for a red light. Looking at the simulation results, shown in Figure 7, there is a mimimum at a segment length of around 200 meter: the simulations were run for a complete tack length of 10 kilometers, and the local minimum is at around 50 track segments.

By mapping the model onto a language with known semantics, we can perform analysis of certain behavioural properties by reusing the semantics and the tooling (in particular, simulators) of that formalism.

4 CONCLUSION

In this tutorial, we presented the use of domain-specific modelling languages for simulation. We motivated the use of such languages in specific domains to improve understanding and to improve the efficiency of domain experts in defining their models, instead of having to use general-purpose languages.

We presented the different aspects of a domain-specific modelling language:

1. abstract syntax to define the allowable constructs;
2. concrete syntax to define the visual representation of abstract syntax constructs;
3. semantic domain to define the domain in which the semantics is expressed;
4. semantic mapping to define the mapping to a model in the semantic domain that defines the (partial) semantics of the domain-specific model.

Each of these aspects was explained and applied to our case study: modelling and analysing the behaviour of a railway network. The behaviour of the railway network is defined using both operational semantics ("simulation") and translational semantics ("mapping", for analysis). A generative approach is used, in order to limit the development overhead.

REFERENCES

Barišić, A., V. Amaral, M. Goulão, and B. Barroca. 2011. "Quality in Use of Domain-specific Languages: A Case Study". In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, 65–72. New York, NY: Association for Computing Machinery.

Cellier, F. E. 1991. *Continuous System Modeling*. New York, NY: Springer-Verlag.

Costagliola, G., V. Deufemia, and G. Polese. 2004. "A Framework for Modeling and Implementing Visual Notations with Applications to Software Engineering". *ACM Transactions on Software Engineering Methodology* 13(4):431–487.

Giese, H., T. Levendovszky, and H. Vangheluwe. 2007. "Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools". In *Models in Software Engineering*, edited by T. Kühne, 252–262. Berlin, Heidelberg: Springer Berlin Heidelberg.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems". *Science of Computer Programming* 8(3):231–274.

Harel, D., and B. Rumpe. 2004. "Meaningful Modeling: What's the Semantics of "Semantics"?". *Computer* 37(10):64–72.

Kelly, S., and J.-P. Tolvanen. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. New York, NY: John Wiley & Sons Inc.

Kleppe, A. 2007, October. "A language description is more than a metamodel". In *Fourth International Workshop on Software Language Engineering*. Nashville, TA.

Kühne, T. 2006. "Matters of (Meta-)Modeling". *Software and System Modeling* 5(4):369–385.

Moody, D. 2009. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". *IEEE Transactions on Software Engineering* 35(6):756–779.

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

Murata, T. 1989. "Petri Nets: Properties, Analysis and Applications". *Proceedings of the IEEE* 77(4):541–580.

Petre, M. 1995. "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming". *Communications of the ACM* 38(6):33–44.

Sendall, S., and W. Kozaczynski. 2003. "Model Transformation: The Heart and Soul of Model-Driven Software Development". *IEEE Software* 20(5):42–45.

Syriani, E., and H. Vangheluwe. 2013. "A Modular Timed Graph Transformation Language for Simulation-Based Design". *Software and System Modeling* 12(2):387–414.

Syriani, E., H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. "AToMPM: A Web-based Modeling Environment". In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, edited by Y. Liu and S. Zschaler, Volume 1115, 21–25. Aachen, Germany: CEUR-WS.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: a Distributed Parallel DEVS Simulator". In *Proceedings of the 2015 Spring Simulation Multiconference*, SpringSim '15, 844–851. San Diego, CA: Society for Computer Simulation International.

Vangheluwe, H. 2008. "Foundations of Modelling and Simulation of Complex Systems". *ECEASST* 10: Graph Transformation and Visual Modeling Techniques 2008.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. San Diego, CA: Academic Press.

Zhang, Y., and B. Xu. 2004. "A Survey of Semantic Description Frameworks for Programming Languages". *SIGPLAN Notices* 39(3):14–30.

AUTHOR BIOGRAPHIES

SIMON VAN MIERLO is a post-doctoral researcher at the University of Antwerp (Belgium). He is a member of the modeling, Simulation and Design (MSDL) research lab. For his PhD thesis, he developed debugging techniques for modeling and simulation formalisms by explicitly modeling their executor's control flow using Statecharts. He is the main developer and maintainer of SCCD, a hybrid formalism that combines Statecharts with class diagrams. His e-mail address is simon.vanmierlo@uantwerpen.be.

HANS VANGHELUWE is a Full Professor at the University of Antwerp (Belgium). He heads the modeling, Simulation and Design (MSDL) research lab. In a variety of projects, often with industrial partners, he develops and applies the model-based theory and techniques of Multi-Paradigm modeling (MPM). His current interests are in domain-specific modeling and simulation, including the development of graphical user interfaces for multiple platforms. His e-mail address is hans.vangheluwe@uantwerpen.be.

JOACHIM DENIL is an Assistant Professor at the University of Antwerp. He is also the AnSyMo-CoSyS-lab core-lab manager within Flanders Make. His research combines aspects from computer science, electronics and simulation to support engineers in the model-based design of cyber-physical systems. His e-mail address is joachim.denil@uantwerpen.be.