10th Int. IEEE Systems Conference (SysCon 2016), April 2016, Orlando, Florida

Creation of Domain-Specific Languages for Executable System Models with the Eclipse Modeling Project

Sven Jäger, Ralph Maschotta, Tino Jungebloud, Alexander Wichmann, and Armin Zimmermann
Systems and Software Engineering Group
Computer Science and Automation Department
Technische Universität Ilmenau
Ilmenau, Germany
Contact: see http://www.tu-ilmenau.de/sse/

***Abstract*—Model-based systems engineering is an increasingly accepted method supporting design decisions. System engineers or modelers have the choice between tools and system description languages that are either abstract and generic or specifically adapted to their domain. The latter approach is easier and more efficient but restrictive. The success of this approach strongly relies on the support of domain-specific tools. The design or adaptation of such software tools and their underlying conceptual models is a complex task, which can be supported by a model-based approach on the meta model level itself.**

This paper proposes a workflow for designing complex systems by using domain-specific models which may combine structural and behavioral aspects. It is loosely based on the Object Management Group's Model Driven Architecture approach. For this purpose we use the Eclipse Modeling Framework and Eclipse Sirius Project, which are part of the Eclipse Modeling Project. The paper describes the complete workflow based on a simple real-life system example to clarify our approach. It covering the design of the domain-specific language, semi-automatic model editor generation, modeling the system, and finally executing a simulation of its behavior.

***Index Terms*—Model-based system design, domain-specific language, simulation, Eclipse Modeling Project, Sirius project, meta model, Ecore**

I. Introduction

Model-based system design and analysis is more and more becoming the solution to handle the analysis and design of complex systems. The methodology has a wide range of applications, ranging from software, small embedded systems, over vehicles all the way to rockets and spacecraft design.

Formal languages like UML [1] or SysML [2] are increasingly used to model systems. These languages (or model families) are domain-independent and thus have a wide range of possible applications and include a lot of generic elements and relations. The domain independence is advantageous and disadvantageous at the same time, especially when a general purpose language is used to model specific systems. Both modelers and domain experts need a lot of knowledge of the language to understand every modeled aspect.

Why should we use domain-specific languages for system modeling? The description of a system in its own terms leads to an improvement of understandability of the model by non-modeling experts (which the system designers and therefore domain experts often are). It also increases the impact of the model in the process of system design because now both modeling and domain experts can easily understand the model. However, an effective use of this approach for modeling systems can only be achieved if tools support the domain-specific language. This often fails because of the huge extra effort for creating or adapting existing tool environments to support the new language. This part is an additional effort to the actual system modeling. Because development of domain-specific languages and models is an iterative process, a lot of changes are expected during the definition process, which increases the overall effort for software development significantly.

It is essential to reduce the amount of software development tasks when we want to use domain-specific modeling to design systems efficiently.

One way to reduce the effort for complex software systems is the model-driven development (MDD) approach and the model-driven architecture (MDA) [3]. They provide a methodology for defining and generating software based on models. Following this method, domain-specific metamodels (i.e., semi-formal descriptions of the model classes themselves) are defined and used to generate software components afterwards. This methodology increases the quality of software by separating different aspects of a software system and reduces failures in the implementation via generation.

Because a domain-specific language of a system can also be seen as a domain-specific metamodel for a modeling software, it is possible to adapt the *MDA* to define system modeling tools, which can then be used to create specific system models.

The eclipse modeling project [4], [5] is a de facto standard in the field of software modeling. It provides a metamodeling framework and thus simplifies the process from defining metamodels to using models in modeling software. It allows, among others, model components and tree editors to be automatically generated from the model. Because a graphical editor for a system model is an essential part, the eclipse modeling project includes the Graphical modeling framework

(*GMF*) and the *Sirius* project [6], [7] respectively its closed source counterpart *Obeo Designer* [8]. Both frameworks use models for describing graphical elements and tools for their manipulation. A comprehension which in includes both frameworks can be found in [9]. The *GMF* uses an approach similar to the eclipse modeling framework, and uses generation for creating editor plugins. The second framework *Sirius* do not need the generation step of *GMF*. It interprets the model of the graphical editor. The capability to interpret the model gives the possibility to directly view the resulting editor while modeling it. This accelerates the development cycles of a modeling tool significantly [8]. The effort for software development is reduced by using these tools, and the designer can focus more on the actual task of system modeling.

The resulting system model can be used in a variety of applications. If an execution semantics is defined, the system model can be simulated and its results can be analyzed. Static analyses are also possible. With the help of generators [10], [11], parts of the system model can be used to generate artifacts for the real system.

Because the ecore metamodel only describe structural behavior, it is necessary to specify an behavioral metamodel. Like in [12] the modeled behavior is often transformed into Java sourcecode which need to be run separately. The resulting java sourcecode is also integrated into the methods to enhance the metamodel with behavior [13]. A more vastly better way is to interpret the modeled, which can easy be done during the design time of the system model and do not need an additional compilation step. For example, this is done by using story diagrams which can be executed by an interpreter in [14].

The contribution of this paper is to propose and demonstrate a workflow for designing executable system models by using domain-specific modeling for combined structural and behavioral system models. With the help of the Eclipse modeling project and the new Sirius project this approach becomes more effective than other approaches. Also an inspection of the simulation is done by using a special viewpoint of the system model.

The paper starts with the description of the methodology in Section II. By using a simple example of a model train system, the structural metamodel and a behavioral metamodel for the system are designed in Section III. The corresponding graphical editor is specified as well. With the help of the graphical editor the system model is designed afterwards. A conclusion is given at the end.

II. Methodology

This section shows our general approach for designing models of systems using the Eclipse modeling project. An overview of the overall design workflow is given in the activity diagram shown in Figure 1.

The workflow is divided in three vertical layers which are based on OMG's layer structure. The OMG defines four different model layers: the meta metamodel layer (M3), the metamodel layer (M2), the model layer (M1) and the instance layer (M0). Each layer uses the elements of the upper layer to describe the elements for modeling the lower layer. This four-layer structure is not fixed for every application. Depending on the specific modeling problem it is also possible to use fewer layers — in our proposal, for instance, we use a three-layer approach.

The topmost layer is the meta metamodel layer which is given by the Eclipse modeling project. It is called *Ecore* and defines the base modeling language which can be used to describe classes, attributes, data types and their relations. Because OMG's *EMOF* was influenced by *Ecore* during the specification of the meta metamodel, both meta metamodels have a similar expressiveness. The meta metamodel layer is followed by the metamodel layer to define the domain-specific language used for creating system models in layer 1. We do not have an instance layer, because the instance of the system model should be a prototype or the real system.

A. Create Metamodel of Behavior

Besides the vertical separation of different modeling layers we also divided the workflow horizontally. The horizontal division visualizes the structural description and the behavioral description of a system. *Ecore* as well as *EMOF* only consider the structural description of a metamodel. To model the behavior of a metamodel as well, we need to create a behavioral metamodel for this layer. This is done in the rightmost part of Figure 1, the *general behavior* by using the meta metamodel elements. The behavioral model is domain-independent and introduces elements which can be used to express behavior on the level of the metamodel. It is possible to use existing behavioral model semantics such as Petri nets or activity diagrams.

By using the Eclipse Modeling Project, the designed behavioral metamodel can be integrated directly into the specifying process of the domain-specific language. Therefore, graphical editors and tree editors can be generated directly from the metamodel without or very little additional effort.

The actual design of the domain-specific language and the system model is done in the two leftmost sections of Figure 1. The proposed workflow has two main inputs, which are influencing the contents of the domain-specific metamodel and the system model layer: the *domain requirements* and the *system requirements*.

B. Model the Domain

The workflow continues with the analysis of the domain requirements and the derivation of the domain-specific language which is represented by the corresponding metamodel. The definition is done by the structured action *Model the Domain* (See Figure 1). In this action, the domain requirements are analyzed and the essential elements are identified. By using the meta metamodel the domain-specific metamodel is specified. This task is divided in two parts, which are depending on each other. To gain a complete metamodel it is necessary to describe both the structure and the behavior of domain elements. Thus the structure of the domain-specific metamodel has to defined first. The description of the domain behavior is done by using

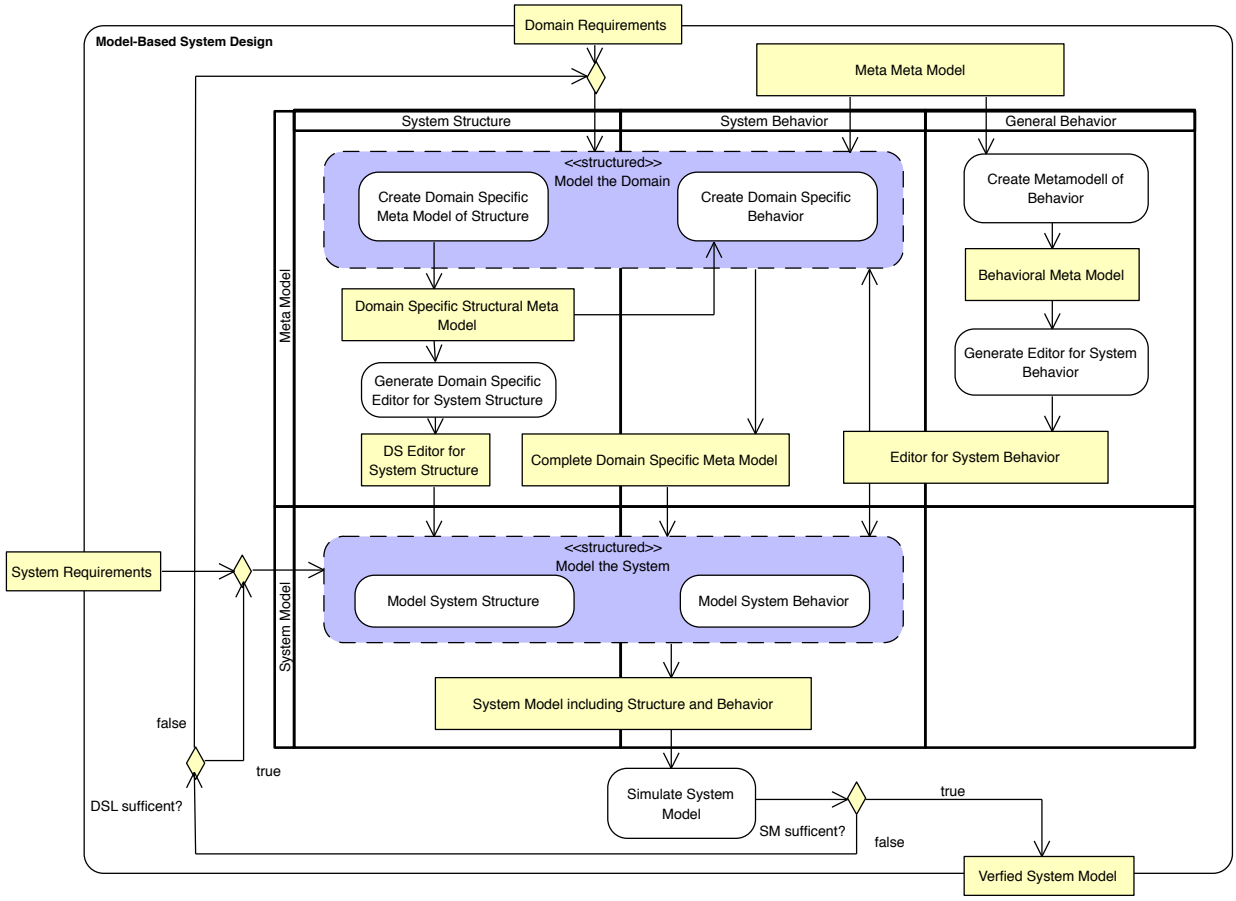

Fig. 1. The activity diagram of our general approach for modeling systems with the eclipse modeling project.

the behavior metamodel of the general behavior. After defining both aspects of the domain-specific model a so-called *complete domain-specific metamodel* can be used to generate the editors for the next layer.

The tasks covered so far are necessary for a modeling and analysis software tool and thus done by a corresponding designer. Based on the result, the following tasks can then be carried out by a system designer with the model class fitting to the domain.

C. Model the System

In the lower layer the actual system modeling is done. In the structured action *Model the System* (See Figure 1) the system requirements are analyzed and used to specify the system model. By using the generated editors for the structural and behavioral aspect of the domain, the system model is described with the elements of the complete domain-specific metamodel. As a result of this action, a system model emerges which includes a structural as well as a behavioral description.

Because the system model also includes the description of the behavior, the system model can be simulated and analyzed after its specification. This requires an *execution semantics* for behavioral elements of the domain - metamodel and the general behavior metamodel.

D. Iterative refinement

The results of the simulation can be used to check if the system model fulfills the system requirements. Otherwise the modeling process goes one layer up to the system modeling layer. At this point the system modeler has to figure out why the system model cannot fulfill the system requirements. There are two possibilities: The first is an insufficient system model. If this is the case, the modeler can fix the existing system model and rerun the simulation. The second possibility could be an inexpressive domain-specific metamodel. This is the case, if it is not possible to express all system requirements in the system model by using the domain specific metamodel. To increase the expressiveness the modeler has to go back to the metamodel layer and adapt the domain model. Such an iteration or adaption is only feasible with a model-based approach like the one proposed here, and will over time for similar projects be enriched and refactored towards an increasingly good modeling base.

This iterative design process can be repeated until the system model fulfills the system requirements. The result of this workflow is a verified system model and can be used to create an instance of this model by creating the real system. The resulting system model can be used for several tasks including:

- design decisions support,
- to find failures in the real system,
- automatically generate software parts,
- prediction of non-observable parameters,
- and optimize the real system.

The success of this iterative system design process approach depends mainly on the effort when switching from one model layer to another model layer. Traditional software development methods would require a lot of work to build the metamodel classes and editors. This was so far done manually or semi-automatically by generating the classes by using a generator. Moreover, it requires an experienced software developer and lot of time to do a model layer switch.

An effective reduction of the effort needed for the software components is required. The eclipse modeling project heavily uses automated model transformation and generation to create metamodel classes and editors. In addition to that, graphical editors can be easily described by using mapping models which map graphical elements to metamodel elements. Such models can then be generated or interpreted by Eclipse. This aspect dramatically reduces the amount of software development effort while switching through the layers, and leads to faster iteration cycles.

III. Example

This section demonstrates the process described above by using a simple example system, namely a simple toy train system used to teach embedded and real-time systems design in our lab. Students have to design a block center to control trains and switches according to a predefined schedule. The positions of the trains can be estimated by using Reed contacts as sensors. The goal is to control two trains at the same time even under unreliable sensor readings, and prevent trains from crashing as well as to avoid deadlocks.

A. General Behavioral Metamodel

The fundamental meta metamodel, the Ecore model, only allows a structural description of the metamodel. By enriching the metamodel with java code, additional behavior can be added. The java code is integrated into the operation body of the resulting Eclipse model plugin when being generated via the Eclipse modeling framework. This is just another way to write source code and has nothing to do with modeling behavior of the system.

In accordance with presented workflow (see section II-A), a simple metamodel-to-model behavior is created, which is shown in Figure 2. It describes the elements of a simple automaton with 4 elements. The root element is the *Automat*. It contains all elements and stores the current position by holding a reference to the active action (*activeAction*). An automaton owns an initial *Context*. It defines the current scope, in which operations are executed. Normally it represents a specific Class. With the help of the *switchReference* and the *switchOperation*, the current context can be switched to another one.

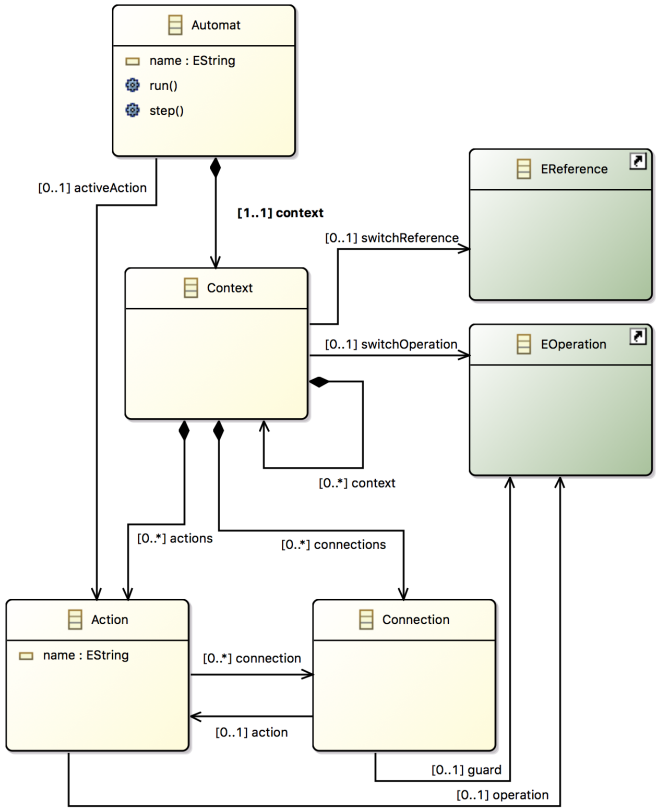

Fig. 2. A simple automaton metamodel which was developed for example system model.

The Context consists of two elements: *Action* and *Connection*. Both elements together can be used to model a directed graph.

Actions are the nodes of the graph. They have a name and can reference to *EOperations* which are invoked when the automaton switches to the action. With the reference *connection* it retrieves the outgoing edges.

A Connection represents an edge of the directed graph. To figure out in which direction the *activeAction* reference has to be moved, it has to execute all *guard* operation of the outgoing connections. These operations return either True or False. If a guard returns True, the activeAction will be set to the action which is referenced via *action*. Because the guard operation is java code, it is not possible to figure out if it is returning True or False (or if it is even returning from execution), especially for complex calculations. The completeness and correctness of the graph can only be analyzed during run time.

The simple execution semantics of this automaton metamodel is shown in Algorithm 1.

The algorithm shows the executed commands for one step. It is included in the *Automaton* class and triggered by the *step()* operation which represents a step execution. The operation *run()* repeats the execution of the function *step()* until the automaton does not change states anymore or arrives at an error state. By adding tracing capabilities to the execution engine, an execution of the automaton can be recorded and

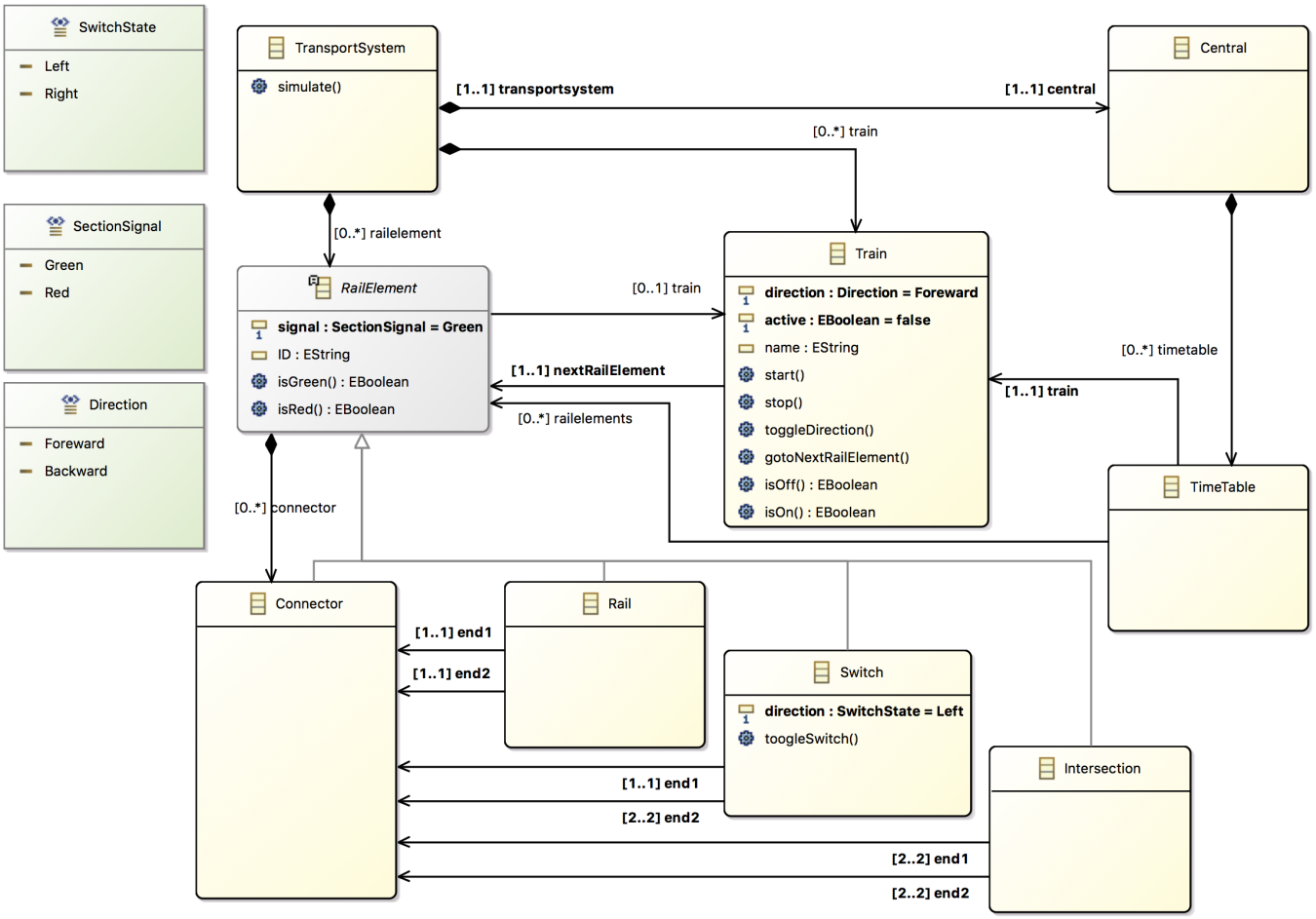

Fig. 3. The structural part of the domain specific metamodel for our example.

Algorithm 1 Algorithm for the execution of an automaton for one step.

```
1:  procedure STEP(activeAction)
2:      for all connection ← activeAction.connection do
3:          context ← connection.getContext()
4:          if context.invoke(connection.guard) then
5:              action ← connection.action
6:              context.invoke(action.operation)
7:              activeAction ← action
8:              return
9:          end if
10:     end for
11: end procedure
```

analyzed later on.

B. Model of the Domain

The next step (see section II-B) is to design the domain-specific metamodel and graphical editors.

1) Structural Metamodel: The structural metamodel defines every structural element in our system. For our railway example it contains all typical structural railway elements. It is described by using the elements of the Ecore metamodel and is shown in Figure 3.

The root element is the *TransportSystem*. It is the container for all elements in the toy railway system and contains rail elements as well as trains and the control center. A part of the rail system is described by a *RailElement*. It is the abstract base class for all elements which represent the structure of the rail system. All rail elements have a signal which shows the allocation status of the rail element. It can be green for free and red for busy. The current state of the rail element can be queried by the function *isGreen()* and *isRed()*. The attribute ID identifies the rail element uniquely. There are three rail elements: *Rails*, *Switches* and *Intersections*. They correspond to actual rail elements of our toy train system. They only differ in the number of *Connectors* and their behavior when a train drives on them. The *Rail* moves a train from the connector on *end1* to one which is referenced by *end2*. A switch moves the train from *end1* to the connectors in *end2* depending on the *direction*, while an *Intersection* moves a train from one connector in *end1* to the opposite connector in *end2*. Each rail element holds at least two *Connectors* to allow the rail element to transfer the train according to its behavior.

The element *Train* represents a train in the transport system. Its state is described by the attributes *direction* and *active*. The direction controls the direction of movement of a train. It can be toggled by using the supplemental function *toggleDirection()*. The engine of train is controlled by the functions *start()* and *stop()*, which change the value of the *active* attribute. The state of engine can be queried by *isOn()* and *isOff()*.

The *Central* is the controller of the transport system. It does not control the train directly but does it indirectly by controlling the signals and the switches of the rail system. By analyzing the *timetable* and the current position of the train, the control center tries to avoid collisions between trains and controls their way through the rail system.

2) Behavior: All supplemental functions in the metamodel either query a state of the model element or change the state. These functions are guards and actions for the description of behavior. They are written in java and implement the domain-specific behavior, which is fixed for the transport system. Only the interesting aspects of the behavior, just like the behavior of the central or the behavior of a train are left open and have to be modeled in the actual rail model.

3) Generation: After the metamodels are defined the meta layer, a translation has to be done. The eclipse modeling framework does this step for us. It derives a generator model from the actual Ecore model and generates the java model classes, the manipulation and creation behavior, and a simple tree editor. Furthermore, it packs everything inside plugins for the Eclipse run-time system.

By registering extensions for the model and editors, Eclipse now "'knows"' our metamodel and we can use it to model the system with a tree editor. Because this step can be done almost automatically, it needs no effort for implementation. Modeling the system is now actual possible, however not yet very comfortable.

4) Graphical Editor: To offer a more user-friendly way to model our system, the next step is to create a graphical editor (user interface). Creating graphical editors normally takes a lot of time and needs a lot of knowledge of graphic engines or frameworks. The Eclipse modeling project provides the *Sirius* environment [6], which reduces the effort for generating graphical editors by using models to define them. The *Sirius* model is called a *viewpoint specification*. Such a viewpoint specification for the structural domain-specific metamodel and the behavior model for our example is shown in Figure 4.

The viewpoint specification model is modeled with the help of a tree editor. Our editor should provide two different views on the model. A *design view*, where the structure of the rail system, the trains and the behavior is designed, and a *simulation viewpoint*, where the current situation is shown during the simulation.

Therefore we define the viewpoints *Design* and *Simulation* in the viewpoint specification. The *Design* viewpoint focuses on the design of the system model. The two specified diagrams *TransportSystem* and *Automat* correspond to our two metamodels for the structural and behavioral description.

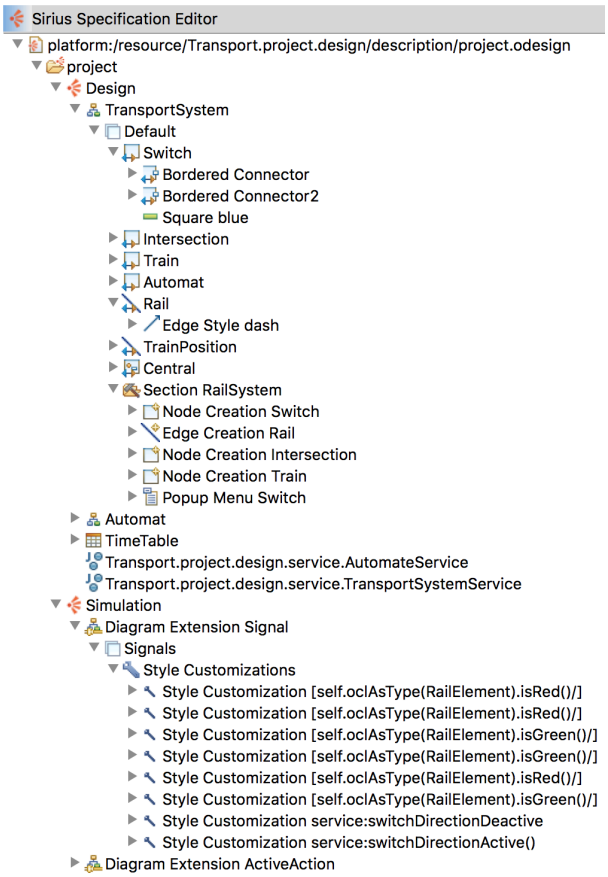

Fig. 4. The tree view of the viewpoint specification model for the graphical editor of the toy train system.

The main components of our system are the nodes *Switch*, *Intersection*, *Train*, and *Central* (for the control center) as well as the *Rail* which is represented by an edge. Each of these nodes define preconditions and selections which are evaluated during run time. If a condition is fulfilled, the Sirius engine adds the graphical element corresponding to the node to the diagram. The Sirius specification introduces an additional element, the *Automat*. This element is the interface between behavioral diagram and structural diagram, and allows to switch easily between diagrams later. To add, manipulate or delete elements in the diagram, the viewpoint specification has to define a toolbox, which is shown under the node *Section RailSystem*.

The diagram node *Automat* specifies the automaton diagram for our system model. The structure corresponds to the behavior metamodel. Besides the specification of diagrams, Sirius can also be used to define other kinds of representations such as tables, cross tables, or sequence diagrams. The *TimeTable* uses the table representation to visualize the sequence of rail elements which a specific train should follow (its schedule).

Supplementary functions which are provided by java classes can be added to increase visual capabilities and to simplify the use of complex behavior inside the editor. An example is shown in the viewpoint specification tree by the *Trans-*

port.project.design.service.TransportSystemService. It simplifies the labeling and includes some additional queries for drawing the active direction of a switch.

The second viewpoint of our system is the *Simulation* viewpoint. It defines the view of the model while the simulation is running. It should be a dynamic view of our system model and visualize the current state. According to the Figure 4, it specifies two diagrams, which are extensions of the *TransportSystem* and *Automat* diagram of the *Design* viewpoint. Both do not introduce new elements.

The signal diagram extension adds a signal layer to the existing diagram. When it is activated during the simulation, the color of each element changes based on the state of the *RailElement*. This is done by using conditional style customization which uses metamodel operations or java service operations to query the states of the *Switches* and the *RailElements*.

Like the signal layer the *ActiveAction* extension change, the state of the automaton is depicted by changing the color of the active action during the execution of the simulation.

With the help of *Sirius* it is thus easy to create a simple graphical designer for an existing meta model, which can have multiple viewpoints and diagrams to model a system and its aspects. A big advantage is the use of automated generation and interpretation, which instantly shows the effect of changes in the diagrams while changing the viewpoint specification. This fits especially for incremental and agile development methods in cases where the full specification or requirements are not known in advance. If special graphical notations inside diagrams are desired, which are not supported out of the box, the effort increases and an experienced software developer becomes necessary.

C. Model of the System

The specification of the editor for the domain-specific metamodel finished the tasks of the metamodel layer of our workflow (see section II). Now the actual modeling of the system model can be created with the editors which were specified in the *viewpoint specification*.

The system model is designed according to our example which was described earlier. It is depicted in Figure 5.

Our rail system contains four *Switches* (blue), one *Intersection* (orange), and eight *Rail* sections (gray). The initial state of the system is shown in the figure, which includes the directions of the switches and the position of the train.

The central represents the control center of our transport system. It contains a schedule for the train *Z1* (for the sake of simplicity, the model is shown for one train only). The two diamonds on the right side are the behavior descriptions for a train and the control center. The symbol on the lower right of the diamond indicates a refinement of the element by another diagram. The diagram can be accessed directly via the context menu of the element.

The refined diagram of the element *Train* is shown in Figure 6. It is a typical automaton graph with states and directed state transitions (edges) connecting them, which is

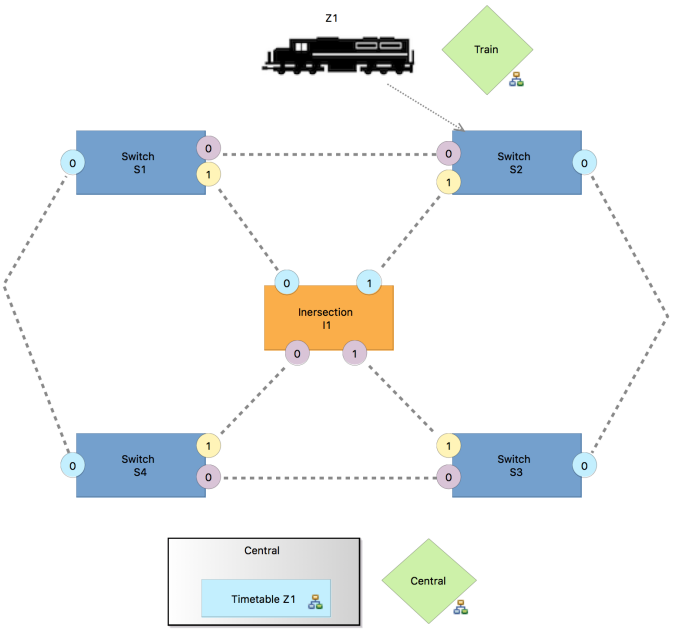

Fig. 5. Structural model of the toy train system.

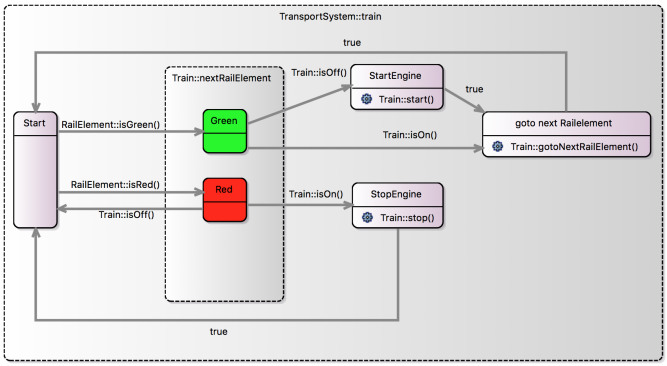

Fig. 6. Automata graph which control the behavior of the train.

used to steer the train from a rail element to the next one. According to the state of the next rail element, the engine of the train is started or stopped. Now have a complete model which combines structural as well as behavioral descriptions for the model train, which is designed inside our example domain-specific model.

D. Simulation

With the help of the defined execution semantics the complete model can finally be simulated to gain knowledge about the correctness, dynamic behavior, or performance of the modeled system. The possibility to observe and analyze the execution of the simulation is part of the execution semantics. It is recommended that it has the capability to trace the execution.

For a better observation it is possible to define simulation viewpoints, which show the current state of the simulation and update during execution. An example for such a diagram is shown in Figure 7.

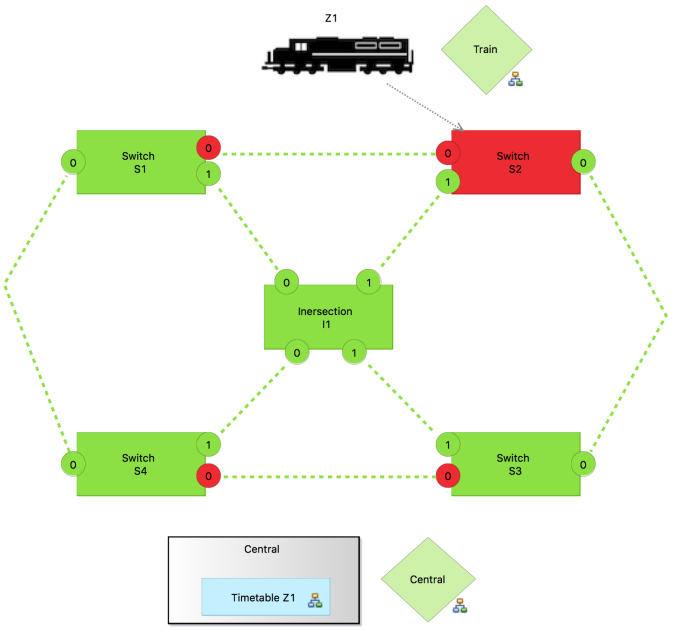

Fig. 7. The signal diagram of the simulation viewpoint for the toy train system during simulation.

It visualizes the state of the signal for each rail element during the simulation. A red element signalizes that an element is allocated by a train, while a green one is currently free. The current direction of the switch can be retrieved by the green connector on the switch. The current train position is shown via an arrow which points to the current *RailElement*. During simulation, this view continuously changes its content according to the changes in states of the depending model elements. Failures in the system model or inaccurate requirements can by detected and corrected with this approach.

IV. Conclusion / Further Work

This paper presented a general workflow for rapidly creating system models by using a domain-specific metamodel to describe structure and behavior of a system. It was shown that the use of the Eclipse modeling project in combination with the recent Sirius project enables this workflow. Significant programming effort is lost in software development for generators and editors otherwise. Now the modeler can focus on designing the metamodel and system model. By using a domain-specific language instead of a general-purpose modeling language, the resulting system model is easier to understand. The workflow was presented by using a small real-life system example from our real-time systems teaching lab. The full set of domain-specific models, graphical editors, system model and simulations only took about three days to design and implement with our proposed approach.

Acknowledgments

This paper is based on work funded by the Federal Ministry for Education and Research of Germany under grant number 01S13031A.

References

[1] OMG, "Unified Modeling Language TM (OMG UML), Version 2.5," Object Management Group, Tech. Rep., June 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/PDF

[2] ——, "Systems Modeling Language (OMG SysML), Version 1.3," Object Management Group, Tech. Rep., 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[3] ——, "Model driven architecture (mda) mda guide rev. 2.0," Object Management Group, Tech. Rep., 2014. [Online]. Available: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[4] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st ed. Addison-Wesley Professional, 2009.

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[6] Eclipse, "Sirius," 2014. [Online]. Available: http://wiki.eclipse.org/Sirius

[7] V. Viyovic, M. Maksimovic, and B. Perisic, "Sirius: A rapid development of dsm graphical editor," in *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, July 2014, pp. 233–238.

[8] E. Juliot and J. Benois, "Viewpoints creation using obeo designer or how to build eclipse dsm without being an expert developer?" Obeo, Tech. Rep., 2010, obeo designer whitepaper.

[9] A. El Kouhen, C. Dumoulin, S. Gerard, and P. Boulet, "Evaluation of Modeling Tools Adaptation," Jan. 2012. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00706701

[10] Eclipse, "Acceleo," 2014. [Online]. Available: http://wiki.eclipse.org/Acceleo

[11] A. S. González, D. S. Ruiz, and G. M. Perez, "Emf4cpp: a c++ ecore implementation," in *DSDM 2010 - Desarrollo de Software Dirigido por Modelos, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2010)*, Valencia, Spain, September 2010.

[12] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr, "eMoflon: Leveraging EMF and Professional CASE Tools," in *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*, 2011.

[13] S. Winetzhammer, "Modgraph - generating executable emf models." *ECEASST*, vol. 54, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/eceasst/eceasst54.html#Winetzhammer12

[14] H. Giese, S. Hildebrandt, and A. Seibel, "Improved flexibility and scalability by interpreting story diagrams," in *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, T. Magaria, J. Padberg, and G. Taentzer, Eds., vol. 18. Electronic Communications of the EASST, 0 2009.