Formalizing train control language: automating analysis of train stations

A. Svendsen[1,2], B. Møller-Pedersen[2], Ø. Haugen[1],
J. Endresen[3] & E. Carlson[3]
[1]*SINTEF, Norway*
[2]*University of Oslo, Norway*
[3]*ABB, Norway*

Abstract

The Train Control Language (TCL) is a domain-specific language that allows automation of the production of interlocking source code. From a graphical editor a model of a train station is created. This model can then be transformed to other representations, e.g. an interlocking table and functional blocks, keeping the representations internally consistent. Formal methods are mathematical techniques for precisely expressing a system, contributing to the reliability and robustness of the system through analysis. Traditionally, applying formal methods involves a high cost. This paper presents a formalization of TCL, including its behavior expressed in the constraint solving language Alloy. We show how analysis of station models can be performed automatically. Analysis, such as simulation of a station, searching for dangerous train movements and deadlocks, is used to illustrate the approach.

Keywords: interlocking, domain specific language (DSL), model analysis, alloy, Train Control Language (TCL).

1 Introduction

An interlocking system prevents dangerous train movement on a train station by giving a "clear" signal to a train only if the requested route is safe. The interlocking system ensures that the route is safe by reading the status of the elements in the route (e.g. tracks, switches, signals) to see if they comply with the logic of the interlocking system. This logic is depicted by an interlocking table, and realized by the interlocking source code, in the form of functional

WIT Transactions on The Built Environment, Vol 114, © 2010 WIT Press
www.witpress.com, ISSN 1743-3509 (on-line)
doi:10.2495/CR100241

blocks of code that are executed (interpreted) by the PLCs (Programmable Logic Controllers) in their control of the station.

Since the interlocking system is a safety system of the highest classification, several rounds of formal review and testing are needed. The functional specification is formally reviewed, before the functional blocks are produced and also formally reviewed in several steps. In addition, systematic testing of the station products is performed to ensure that they are correct. Both the review and testing processes are time-consuming and have a high cost.

The Train Control Language (TCL) [1, 2] is a domain-specific language (DSL) for modeling train stations. TCL automates the production of functional specification and interlocking source code. From a graphical editor, where train stations can be modeled, model transformations generate other representations of the stations, e.g. interlocking tables, functional specifications and functional blocks of interlocking source code.

In this paper we present an extension to our original TCL to automate analysis of train station models. The contribution is the formalization of the TCL language and models, and the analysis performed on these models. Even though the current review and testing processes cannot be eliminated, allowing for automatic analysis on model level may allow reduction of costs in these activities.

The outline of the paper is as follows: Section 2 describes the background for this work, the current development techniques, including the review and testing activities. Section 3 introduces TCL and how it automates the production of interlocking source code. Section 4 briefly introduces the constraint-solving language Alloy that will be used for formalizing TCL in Section 5. Section 6 illustrates how the formal Alloy models can be used for automatic analysis of the TCL models. Finally, Section 7 concludes the paper and look at some topics for future work.

2 Background

From an input requirement specification, consisting of an interlocking table, a structured drawing of the station and a generic Computer Based Interlocking (CBI), incorporating national rules, a functional specification is produced. The functional specification is a mapping of the interlocking table into a set of logical equations. The functional specification is further developed into a design specification, which is close to the interlocking source code. The functional specification and design specification are formally reviewed following the Fagan inspection method [3]. This method includes a set of rules, guidelines and checklists for use in ABB RailLock. Both the production and review of the functional specification and design specification are performed manually, and are thus of high cost.

Following the functional specification and the design specification two teams develop the interlocking source code using different libraries and developing methods. This reduces the chance for common code errors. A formal review of the produced interlocking source code, checking it against the functional

specification and design specification, is then performed using the Fagan inspection method once more. An independent party then validates the source code against all safety requirements using a formal mathematical method that is accepted as adequate by the Norwegian Railway Authority.

Following the review of the interlocking source code, the source code is deployed and several steps of systematic testing are performed. This includes testing the response of the elements in the station, to ensure that they give the correct responses, and simulating train movement systematically, to verify that the system behaves as expected. The behavior of the interlocking system is described by the dynamic semantics of this system, and we model a set of dynamic semantic rules for the interlocking system in Section 5.

3 Train control language

Since the development of interlocking source code is a time-consuming process requiring a large amount of resources, the Train Control Language has been developed to automate this task. This was shown by [1, 2], and in this section we show a summary of this work.

TCL is a domain-specific language for modeling stations in the train domain. TCL is defined by a metamodel (see Figure 1), which defines the concepts in the language and how they are connected.

The topmost concept is *Station*, which represents the station, containing the other concepts. A *TrainRoute* is the route a train must acquire to be allowed to move into or out of the station. A TrainRoute consists of several *TrackCircuits*, which are a collection of *Tracks*, where a train can be detected. A Track can

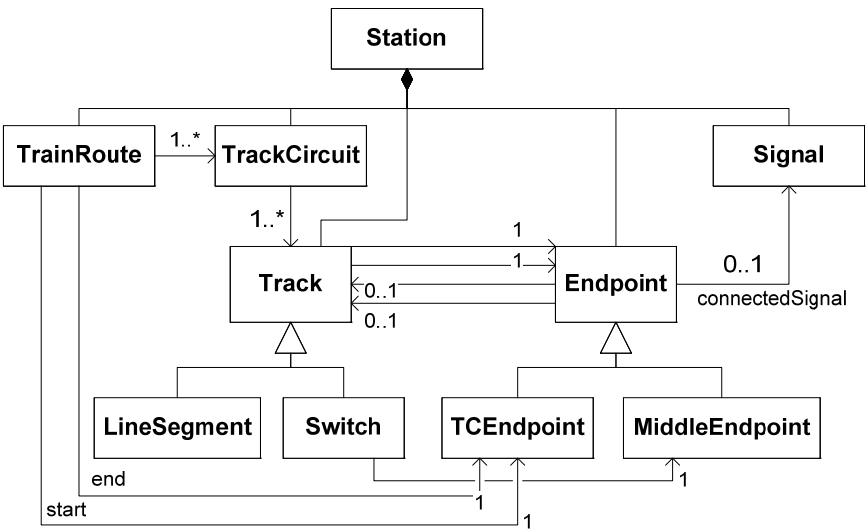

Figure 1: TCL metamodel excerpt.

either be a *LineSegment* or a *Switch*, and these are connected by *Endpoints*. An Endpoint can either divide TrackCircuits (*TCEndpoint*) or be within a TrackCircuit to connect LineSegments and Switches (*MiddleEndpoint*). A TrainRoute starts at a TCEndpoint with a connected *Signal* and ends at another TCEndpoint with a connected Signal in the same direction.

Based on the metamodel, Eclipse Modeling Framework (EMF) [4] and Graphical Modeling Framework (GMF) [5] have been used to develop editors, in particular a graphical editor for modeling the structure of a train station (see Figure 2). The figure also illustrates the concrete syntax of TCL by showing a station with two tracks. A station is created by choosing an element on the toolbar (to the right), dragging it into the canvas (middle) and connecting it to the other elements. Attributes for the elements are then set according to its property (property view at the bottom). When the station model is complete according to the input specification, other representations can be generated automatically by pressing a button (on top).

TCL includes three kinds of model transformations, generating one of the three following representations: Interlocking table, functional specification and interlocking source code (functional blocks). The interlocking table is used to compare with the provided interlocking table to visually verify the correctness of the station in an early phase. The functional specification is also used for

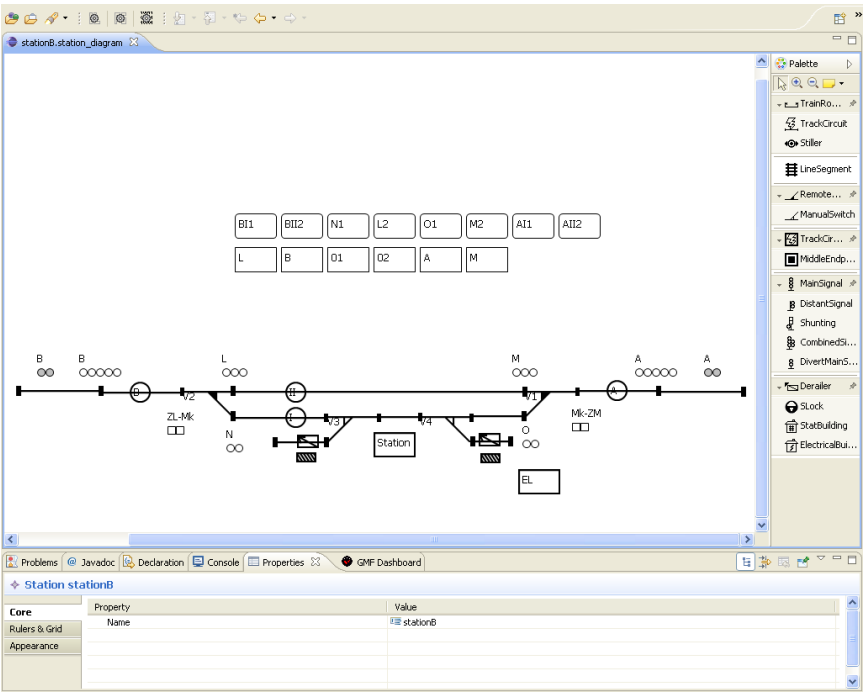

Figure 2: TCL graphical editor.

verification purposes. The interlocking source code has to be formally verified and tested before it can be used for controlling the station.

Notice that for TCL to be put into full production, a formal verification of the language and code generators is needed, to formally confirm that it complies with the same safety standards as the current development process.

We will, however, see how analysis can be performed on the TCL models automatically by translating the models into models of the constrain-solving language Alloy. Since the train domain has to follow high safety standards, this will not eliminate the time-consuming process of reviewing and testing the station products. However, by performing analysis on model level early in the development phase, both design and implementation errors can be discovered early and thus reducing cost.

4 Alloy

Formal verification and validation involves expressing a system (e.g. a train station) precisely through mathematical terms and proving the correctness of the system. Formal methods have traditionally provided accurate analysis of systems at a high cost. Extensive knowledge of mathematical techniques, with their complex notations and theorem proving raises the threshold for performing analysis.

Alloy is a lightweight declarative constraint-solving language for relational calculus [6]. Through the Alloy Analyzer automatic and incremental analysis can be performed without the need for proving theorems or handling complex mathematical notation. Unlike traditional theorem proving, the Alloy Analyzer only offers analysis within a given scope, which is the number of instantiated elements of each type. The small scope hypothesis ensures that such analysis is sufficient, since if a solution exists, it will be within a scope of small size [7].

An Alloy model typically consists of *signatures* (types), *fields* (references to signatures), *facts* (global constraints), *predicates* (parameterized constraints) and *assertions* (claims). A type hierarchy is modeled by letting a signature extend another signature. A fact consists of constraints that must always hold. A predicate consists of constraints that must hold if the predicate is processed, and can therefore be used to represent operations. An assertion consists of constraints that is claimed to hold. As an example, Figure 3 shows a signature of a train route corresponding to train route in the TCL metamodel (Figure 1).

```
abstract sig TrainRoute {                          <-- Signature of train route
  trackCircuits: some TrackCircuit,                <-- Fields referring other signatures
  start: one TrackCircuitEndpoint,
  end: one TrackCircuitEndpoint,
  direction: one Direction
}{one st:Station | this in st.trainRoutes}         <-- Fact: Train route contained by Station

//trainroutes have to refer different endpoints
fact {no t:TrainRoute, e:t.start, e2:t.end | e = e2}<-- Fact: Train routes start and end at different places
```

Figure 3: Signature of a train route in Alloy.

In the search for a solution, the Alloy Analyzer populates the signatures with elements up to the given scope where all the facts are satisfied. Two kinds of analysis can be performed: Finding a model instance satisfying a predicate or finding a model, which represents a counterexample to an assertion. If an analysis does not find a solution or counterexample, there may not be any solution or counterexample within the selected scope, or the constraints (facts/predicates) may over-constrain the model. Thus, the constraints can be adjusted and the Alloy model can be built stepwise based on the feedback from the Alloy Analyzer.

The Alloy Analyzer requires the maximum number of each type of element (scope) to be specified, and it guarantees that if a solution or counterexample exists within the scope, the analysis will find it. This process does not require any test cases, since it checks a property for all possible solutions within the scope. The space of cases examined by the analysis is usually huge (billions of cases) [6].

5 Formalizing TCL

For the formalization of TCL we follow the approach by Kelsen and Ma [8]. They illustrate how to use Alloy to formalize modeling languages and compare it to traditional formalization techniques. As they point out, the Alloy approach offers a uniform notation and automatic analyzability using the Alloy Analyzer.

We choose to formalize TCL in Alloy by three separate models; a static model, a dynamic model and an instance model (see Figure 4). Semantic rules on language level can then be separated from the rules on instance level, such that several instances can use the same static rules. Besides that, we get a clear separation between static and dynamic semantics, making them easier to maintain.

The static model holds the static semantics for the TCL language, including the concepts and how they relate (from the metamodel) in addition to language constraints. Figure 3 shows how the concept train route is modeled in Alloy by using a signature. This signature relate to other signatures through its fields (e.g. to track circuit and endpoints). Additional constraints restrict the number of valid TCL models instantiated by the Alloy Analyzer.

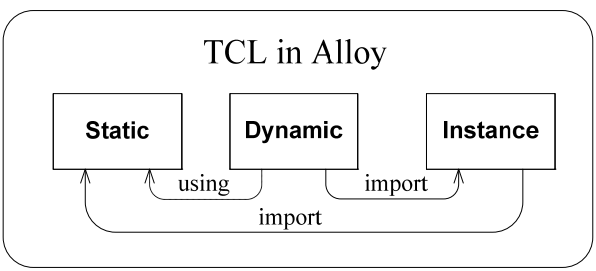

Figure 4: Alloy specification divided into three models.

The Alloy Analyzer populates the signatures with elements when it searches for solutions or counterexamples. Thus an arbitrarily TCL model is instantiated when using the static model. However, since we want to analyze a particular TCL model created by the TCL editor, the number of valid model instances in Alloy must be further constrained to be only this one. We therefore *import* and extend the static semantics of the static model using an instance model, which specifies one particular station. The instance model therefore specifies the number of model elements in the TCL model and how they are connected (e.g. the exact number of train routes and how they are connected to other elements). The result of these constraints is that Alloy only instantiates one valid model for the analysis, which is the TCL model subject to analysis.

To be able to perform proper analysis on a TCL model, the behavior of the station needs to be formally specified as well. This specification is modeled in the dynamic model using a state machine. The dynamic model constrains the behavior of the concepts of the static model, and the Alloy Analyzer satisfies these constraints when it uses the instance model to instantiate an instance. Therefore, the dynamic model *imports* the instance model and *uses* the concepts of the static model.

The dynamic semantics of TCL involves train movement. Intuitively, trains can move simultaneously on a station as long as they follow the basic rules of the interlocking table (table defining safe train movement). More specifically, a train has to request a train route before it can move into or out of the station. Given that no other conflicting routes are already taken and all track circuits in the route are free, the route can be given to the train. The allocation of the train route involves setting switches to the right position and signals to the correct status before the train gets a "clear" signal. The train moves from track circuit to track circuit within the route until it reaches its destination. The track circuits are occupied and freed during the movement.

The state machine defined in Alloy, to describe the behavior of a station, contains a set of states and trains in addition to the instance of the TCL model. The states define the conditions of the station (e.g. position of trains) and the transitions between them define the operation to be performed. There are three operations (represented as predicates): Insert a new train on either side of the station, allocate a route to a train, and moving a train. Through these three operations we can simulate the train movement on the TCL model modeled by the TCL editor.

The development of the Alloy models is illustrated in Figure 5. The static and dynamic models are defining the TCL language and are thus only produced once. The static model is generated from the TCL metamodel, while the dynamic model is produced manually. The instance model is different for each TCL model, and is therefore generated once for each TCL model. However, the instance model is generated automatically from the TCL model modeled in the TCL editor using a MOFScript transformation [9].

As a comparison, Jackson presents an Alloy case study on railway safety [10]. In this example constraints are specified such that only safe train movement is allowed. This is very similar to our Alloy approach. However, our approach

performs analysis on real train stations, which are typically more complex than the example presented in [10].

6 Performing analysis of TCL models

From the Alloy formalization of TCL we can perform analysis on the TCL models. The Alloy Analyzer can, as mentioned in Section 4, perform two kinds of analyses: searching for a solution that satisfies a predicate or searching for a counterexample that falsifies an assertion. In our analysis we will use both of these to prove certain properties.

To perform analysis on a TCL model, the TCL model is exported and transformed to an Alloy instance model (as described in Section 5) and the Alloy Analyzer is invoked with this model as input. This process has been integrated into the TCL editor giving a user-friendly interface for performing the analysis on TCL models. Figure 6 illustrates the integration with the TCL editor, and how

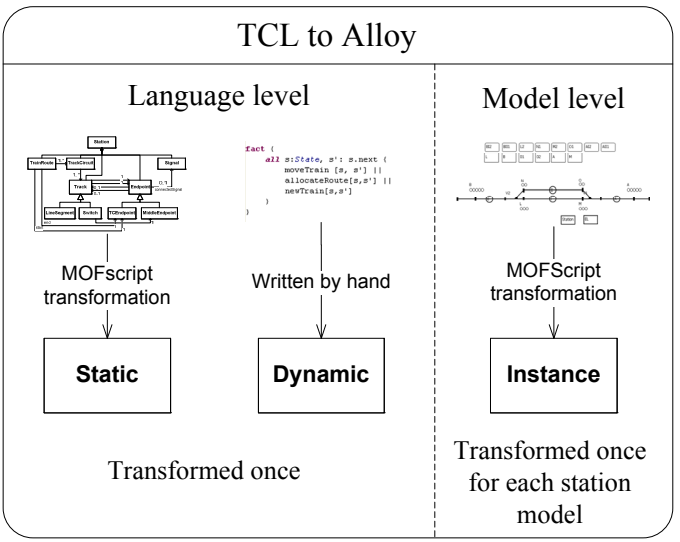

Figure 5: Development of the Alloy models.

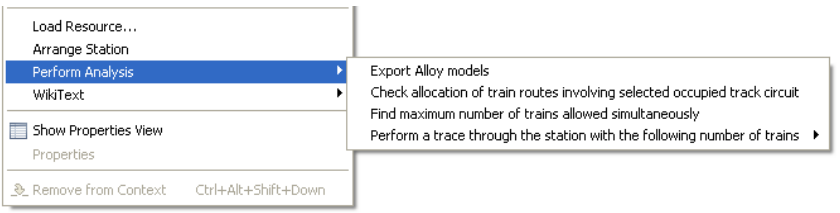

Figure 6: Integration with the TCL editor.

to perform the analysis. By right clicking on the station canvas, the illustrated menu is given where one of the menu items can be selected. Only a few options for analysis are included in this interface for now. However, we plan to add more options in the future, including a possibility to specify arbitrarily predicates and assertions.

Our analysis is mainly concerned with the behavior of the station (dynamic semantics) in some particular situations. The Alloy Analyzer gives a solution or counterexample by giving a trace through the state machine specified by the dynamic semantics. By following this trace, we can observe how the condition of the station changes, and thus see the train movement through the station. Constraints for the conditions in the first and last state in the trace can be specified (e.g. both the first and last state includes no trains in the station).

Intuitively, we specify the conditions for the first state and for the last state in the trace and how many trains are moving through the station. These parameters, in addition to whether we run a predicate or check an assertion, decide what kind of analysis we are performing.

As an example, imagine that we have a start condition with a train on track 1 (see Figure 7). Typical test-cases will be to test whether any train routes involving track 1 (train route 1 and 2 in Figure 7) can be given to other trains while the train is located on track 1. This property can be checked through specifying an assertion in Alloy (see Figure 8). This assertion claims that no model can be instantiated where the following constraints are true: The first state in the trace includes a train on track 1, the last state in the trace still constrains the train to be on track 1, and the last state in the trace also includes an allocated route (to another train) involving track 1. The Alloy Analyzer is invoked to find a possible trace through the state machine where such behavior is allowed (a counterexample). Fortunately, for our two-track station, Alloy does not find any counterexample that falsifies our assertion, proving that no train routes involving track 1 can be allocated when a train is located there.

Other analyses include the search for the number of active trains the station can include simultaneously without leading to a deadlock. A predicate can be used to search for a solution for a certain number of simultaneous trains. If no solution is found, the specified number of simultaneous trains will lead to a deadlock. For our two-track station, the maximum number of simultaneous trains turns out to be three (solution illustrated in Figure 9). This figure illustrates the

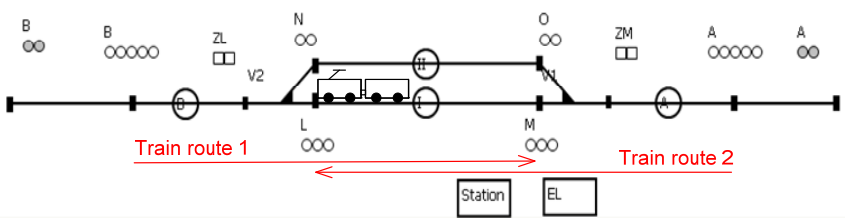

Figure 7: Two-track station with a train on track 1.

```
assert checkRouteAllocation {

      //assert that no model with the following constraints exist
      no t,t2:Train, tc:tc_01, tr:TrainRoute{
            tc in tr.trackCircuits

            //constraints for first state in trace
            t->tc in first.trainOnTrack
            tc in first.occupiedTrack
            no first.trainOnRoute
            no first.allocatedRoute

            //constraints for last state in trace
            t->tc in last.trainOnTrack
            tc in last.occupiedTrack
            t2->tc in last.trainOnTrack
            t2->tr in last.trainOnRoute
            tr in last.allocatedRoute
      }
}
```

Figure 8: Assertion on train route allocation involving track circuit 01.

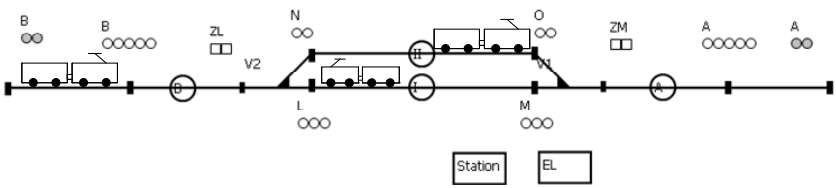

Figure 9: Maximum number of trains on the station simultaneously.

condition of the station in the state (in the trace) where it included three trains simultaneously. Notice that this figure has been created based on the trace information given by the Alloy Analyzer, and is not created by Alloy itself.

Arbitrarily analysis can thus be performed automatically by specifying the condition of the first and last states in the trace, the number of trains to be involved and what kind of assertion/predicate to check/process. We have seen two examples of analysis that can be performed on a TCL model. However, we see that these two examples do not differ from other test cases on stations. Thus, a big amount of the testing of stations can be similarly checked by analyzing the TCL models, with considerable less amount of effort.

7 Conclusion and future work

This paper presented a formalization of TCL, both static and dynamic semantics, in Alloy such that automatic analysis can be performed on TCL models. We looked at how the process of performing this analysis has been simplified by

integration with the TCL editor. Furthermore, two examples of analysis were presented to illustrate the approach.

As pointed out, this approach may not replace the traditional validation, verification and testing processes. However, it adds extra value by allowing automatic analysis in the early development process, which can be performed both in the designing phase and in the development phase. By simulating train movement traces on different station architectures (models), errors can be discovered and corrected early, making a considerable potential for reducing cost and time-to-market.

Furthermore, since this approach analyzes TCL models, it will shift the necessity of validation and verification from the code level to the model transformations. However, validation and verification of the model transformations only needs to be performed once. This approach thus has a huge potential of optimizing the development and testing of interlocking source code.

As future work we plan to extend the analysis we perform on TCL models. Since the analysis is performed automatically, we can easily extend it to include other test cases and properties that were earlier checked manually. Furthermore, we are currently working on verifying the interlocking source code generated by the TCL code generators. With verified code generators, parts of the verification and testing process can be performed automatically on model level.

Acknowledgements

The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 part of the Eureka framework.

References

[1] Endresen, J., et al. Train control language - teaching computers interlocking. in Computers in Railways XI (COMPRAIL 2008). 2008. Toledo, Spain: WIT Press.
[2] Svendsen, A., et al. The Future of Train Signaling. in Model Driven Engineering Languages and Systems (MoDELS 2008). 2008. Tolouse, France: Springer.
[3] Fagan, M.E., Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, 1976. **15**(3): p. 182-211.
[4] EMF, Eclipse Modeling Framework (EMF): http://www.eclipse.org/modeling/emf/.
[5] GMF, Eclipse Graphical Modeling Framework (GMF): http://www.eclipse.org/modeling/gmf/.
[6] Jackson, D., Software Abstractions: Logic, Language, and Analysis. 2006: The MIT Press.
[7] Andoni, A., et al., Evaluating the “Small Scope Hypothesis”. 2003, MIT CSAIL.
[8] Kelsen, P. and Q. Ma, A Lightweight Approach for Defining the Formal Semantics of a Modeling Language, in Proceedings of the 11th

international conference on Model Driven Engineering Languages and Systems. 2008, Springer-Verlag: Toulouse, France.
[9] Oldevik, J., MOFScript Eclipse Plug-In: Metamodel-Based Code Generation, in Eclipse Technology Workshop (EtX) at ECOOP 2006. 2006: Nantes.
[10] Jackson, D., Micromodels of Software, in Models, Algebras and Logic of Engineering Software, M. Broy and M. Pizka, Editors. 2003, IOS Press. p. 351-384.