SINTEF

SINTEF ICT

Address: P.O.Box 124, Blindern
0314 Oslo NORWAY
Location: Forskningsveien 1
0373 Oslo
Telephone: +47 22 06 73 00
Fax: +47 22 06 73 50

Enterprise No.: NO 948 007 029 MVA

SINTEF REPORT

TITLE

Improving Domain-Specific Languages by Analyzing, Constraining and Enhancing Metamodels

AUTHOR(S)

Andreas Svendsen, Øystein Haugen, Birger Møller-Pedersen

CLIENT(S)

Research Project MoSiS and Research Project VERDE

REPORT NO.	CLASSIFICATION	CLIENTS REF.	
SINTEF A21093	OPEN	Research Council of Norway project nr. 180110/I40 and 193264/I40	
CLASS. THIS PAGE	ISBN	PROJECT NO.	NO. OF PAGES/APPENDICES
OPEN	978-82-14-04995-4	90B246 and 90B274	25
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.)	CHECKED BY (NAME, SIGN.)
N/A		Øystein Haugen	Arnor Solberg
FILE CODE	DATE	APPROVED BY (NAME, POSITION, SIGN.)	
N/A	2011-11-10	Bjørn Skjellaug, Research Director	

ABSTRACT

We present an approach for improving domain-specific modeling languages (DSML) by automatically revealing unintended models and subsequently introducing constraints to disallow these. One purpose with domain-specific modeling is to raise the level of abstraction by restricting application models to be within a domain. A metamodel, describing the concepts of the language, will typically restrict the type of concepts and how they are connected. However, these restrictions are not sufficient since the number of possible illegal models can still be large. Using a formal definition of the static semantics, we generate arbitrary models of a DSML. Based on these models, we show how to incrementally constrain the language to prohibit unintended models. We provide a prototype implementation of the approach, and we apply this prototype to an example in the train domain to illustrate the approach.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	Modeling	Modellering
GROUP 2	Domain-Specific Modeling	Domene-spesifikk modellering
SELECTED BY AUTHOR	Metamodeling	Metamodellering
	Alloy	Alloy
	Train Control Language	Train Control Language

Improving Domain-Specific Languages by Analyzing, Constraining and Enhancing Metamodels

Andreas Svendsen[1,2], Øystein Haugen[1], and Birger Møller-Pedersen[2]

[1]SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
[2]Department of Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
andreas.svendsen@sintef.no, oystein.haugen@sintef.no, birger@ifi.uio.no

Abstract. We present an approach for improving domain-specific modeling languages (DSML) by automatically revealing unintended models and subsequently introducing constraints to disallow these. One purpose with domain-specific modeling is to raise the level of abstraction by restricting application models to be within a domain. A metamodel, describing the concepts of the language, will typically restrict the type of concepts and how they are connected. However, these restrictions are not sufficient since the number of possible illegal models can still be large. Using a formal definition of the static semantics, we generate arbitrary models of a DSML. Based on these models, we show how to incrementally constrain the language to prohibit unintended models. We provide a prototype implementation of the approach, and we apply this prototype to an example in the train domain to illustrate the approach.

Keywords: Metamodel, domain-specific language, Alloy, Train Control Language.

1 Introduction

Domain-Specific Modeling (DSM) has recently increased in popularity, since it allows the developer to abstractly express a problem within a given domain by making models in a language that is specific for the domain. It has been shown that DSM techniques can increase efficiency of developing software systems [15]. A system within the domain is modeled using a Domain-Specific Language (DSL), which is tailored to the domain. The concepts of the domain are captured by a metamodel, which can be instantiated as models in this DSL. Even though DSLs usually are quite small, designing a DSL requires several iterations of improvement. Is it possible to have these steps of improvement assisted by analysis tools?

Since the language is specific to a domain, we should only be allowed to define meaningful models within the domain. However, the metamodel may not be restrictive enough, and can consequently allow unintended models. On the other hand it may be too restrictive, disallowing intended models. Identifying these weaknesses of a metamodel is normally a manual and tedious process. In this paper we propose an approach for revealing unintended models and identifying weaknesses of the

corresponding metamodel. We use the Alloy Analyzer to generate solution models, which are transformed to DSL models corresponding to a metamodel. These models can reveal weaknesses of the metamodel in the form of unintended models. We then discuss how to enhance and constrain the metamodel according to the identified weaknesses. The approach is illustrated using a concrete example from the train domain, where we incrementally constrain and enhance the metamodel in question. Furthermore, the approach is realized by an implementation based on Eclipse and Ecore, where Ecore metamodels are instantiated and subsequently constrained and enhanced. Fig. 1 gives an overview of the approach, where we constrain the metamodel to prohibit the possibility of creating unintended models.

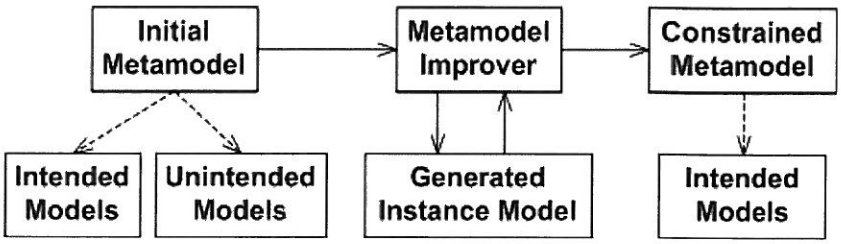

Fig. 1. Overview of the approach

The contribution of this paper is as follows: We give an approach for identifying unintended models by automatically generating models corresponding to a metamodel. We provide a mapping between an Ecore metamodel and Alloy, and consequently a generic generation of an Alloy model from a metamodel. Furthermore, we present how to generically produce a model in a DSL (that is a model according to the metamodel of the DSL) from an Alloy solution model. In addition we discuss how to identify the weaknesses of a metamodel, and subsequently how to enhance and constrain it. A prototype implementation of the approach, including the translation of constraints between Alloy and the Object Constraint Language (OCL), is also discussed.

The outline of the paper is as follows: Section 2 gives background information about DSM, the train DSL, Alloy and OCL. Section 3 describes the approach and illustrates it on an example (train control DSL), while Section 4 gives details about the Ecore implementation. Section 5 discusses weaknesses and strengths about the approach, and also gives some added value of dynamic analysis. Section 6 presents related work, and finally Section 7 gives some concluding remarks and future work.

2 Background

Before describing the approach, we first give background information to explain the concepts and technologies used in this paper. We give details about domain-specific modeling and how domain-specific modeling languages (DSML) are specified by metamodels. Even though our approach is general, the discussion and implementation use the Eclipse platform with Ecore [6] as a base. Then we introduce the example from the train domain, which will be used throughout the paper. This is followed by an introduction to Alloy and how it can be used to perform automatic analysis by

means of its Analyzer tool. Since we use Ecore as a base when enhancing and constraining the metamodel, we give an introduction to the Object Constraint Language (OCL) [19].

2.1 Domain-Specific Modeling

Domain-specific modeling is a software engineering methodology for using models to specify applications within a particular domain. The purpose of DSM is to raise the level of abstraction by only using the concepts of the domain and hiding low level implementation details. This is achieved by using three artifacts: A DSL, a code-generator and a domain framework [15]. A DSL typically defines concepts and rules of the domain using a metamodel for the abstract syntax, and a (graphical) concrete syntax that resembles the way phenomena in the domain usually are depicted. Models in the DSL are input to the code-generator, which generate product code according to the domain framework.

A metamodel is a model which defines the abstract syntax of a DSL, which corresponds to the concepts in a domain and how they are related. An example of a metamodel is illustrated in Fig. 2a, where the concepts *A, B* and *C* are represented by class *A, B* and *C,* respectively. These classes relate to each other by containment or associations. Metamodels can be instantiated into models, and these models are then models of applications in the domain. A model usually consists of two parts: The model element objects in form of repository elements (abstract syntax tree) and the graphical notation (concrete syntax). We will later see that we use both of these in the example. Fig. 2b illustrates the model hierarchy described here.

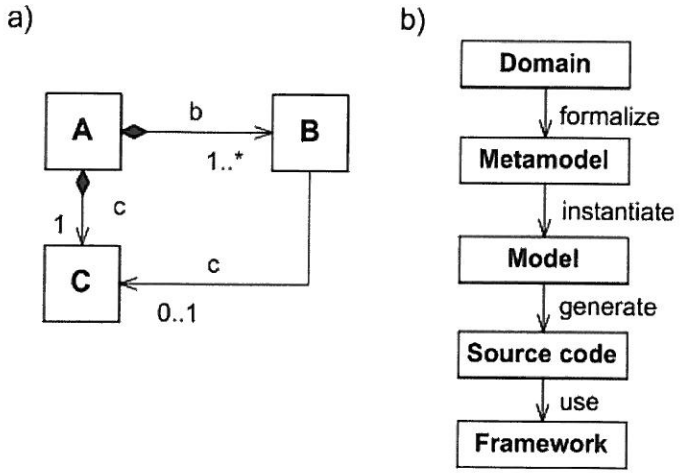

Fig. 2. Metamodel example and model hierarchy

There are several DSM environments available. Since the Eclipse Modeling Framework (EMF) [6] is an open-source platform, this report will use EMF in the discussion and example. However, the approach is not limited to the EMF platform.

2.2 Train Control Language

Train Control Language (TCL) is a Domain-Specific Modeling Language (DSML) for modeling signaling systems on train stations. These are safety-critical systems, and the purpose of TCL is to automate the production of interlocking source code. This code ensures that only safe train movement is allowed. TCL raises the level of abstraction by defining a graphical concrete syntax that is already familiar for domain experts and thus hiding the implementation details. Domain experts without programming or modeling experience can therefore design station models and generate code from these station models.

TCL is defined by an Ecore metamodel, which captures all the significant concepts of a station and how they are related (see Fig. 3). The topmost element in the TCL metamodel is *Station*, which represents the station, containing the other elements. A *TrainRoute* is the route a train must acquire to be able to move into or out of the station. A TrainRoute consists of several *TrackCircuits*, which is a group of *Tracks*, where a train can be located. A Track can either be a *LineSegment* or a *Switch*, and these are connected by *Endpoints*. An Endpoint can either divide two TrackCircuits (*TrackCircuitEndpoint*) or be within a TrackCircuit (*MiddleEndpoint*). A TrainRoute starts at a TrackCircuitEndpoint with a connected *MainSignal* and ends at another TrackCircuitEndpoint with a connected MainSignal in the same direction. The concrete syntax of TCL is illustrated in Fig. 4. In addition TCL contains *Stillers*, *Buildings*, other kinds of *Signals* etc. Note that Fig. 3 does not show attributes, data types, and only a few association names. We refer to [27] and [7] for more information about the details of TCL.

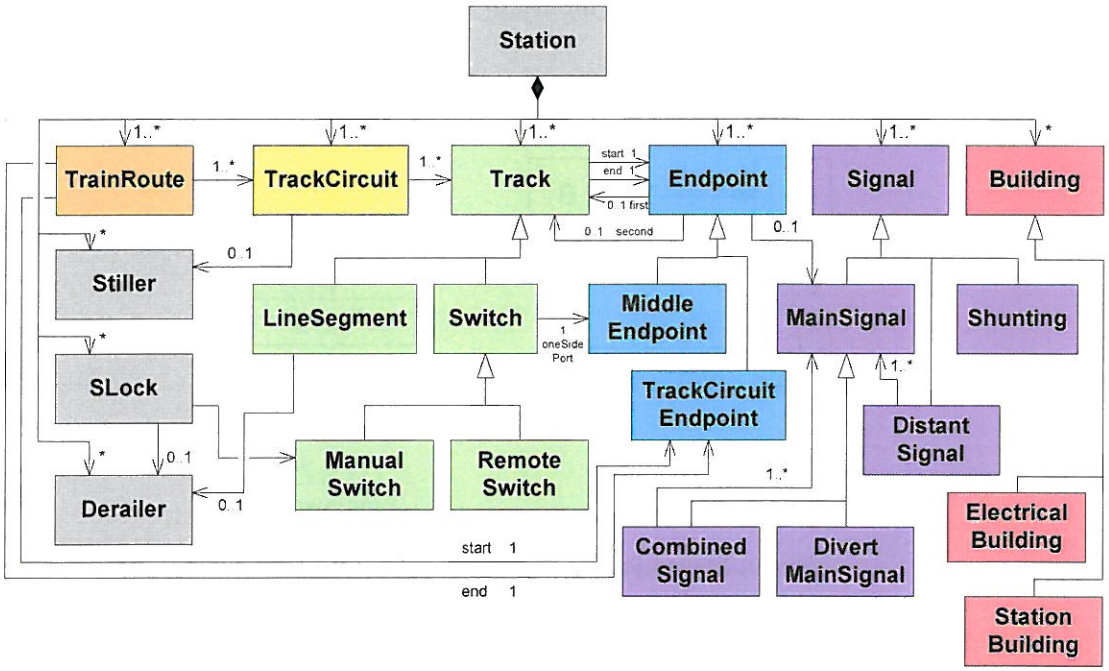

Fig. 3. TCL metamodel excerpt

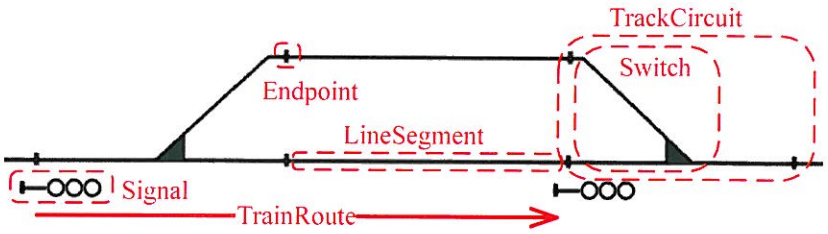

Fig. 4. The concrete syntax of TCL annotated by TCL concepts

The tool suite for TCL includes a graphical editor, code generator and a model analyzer to support the modeling of train stations and generation of interlocking source code. An illustration of the graphical editor with a model of a two-track station with a side-track is shown in Fig. 5. This figure illustrates how a typical station looks like. Note that the rectangles on top represent the TrainRoutes and TrackCircuits. The TCL model analyzer is based upon a formal description of both the static and dynamic semantics of TCL in Alloy. Dynamic simulation and checking of properties of a station model are performed automatically by the Alloy Analyzer. For further information about the analysis of TCL models, we refer to [26].

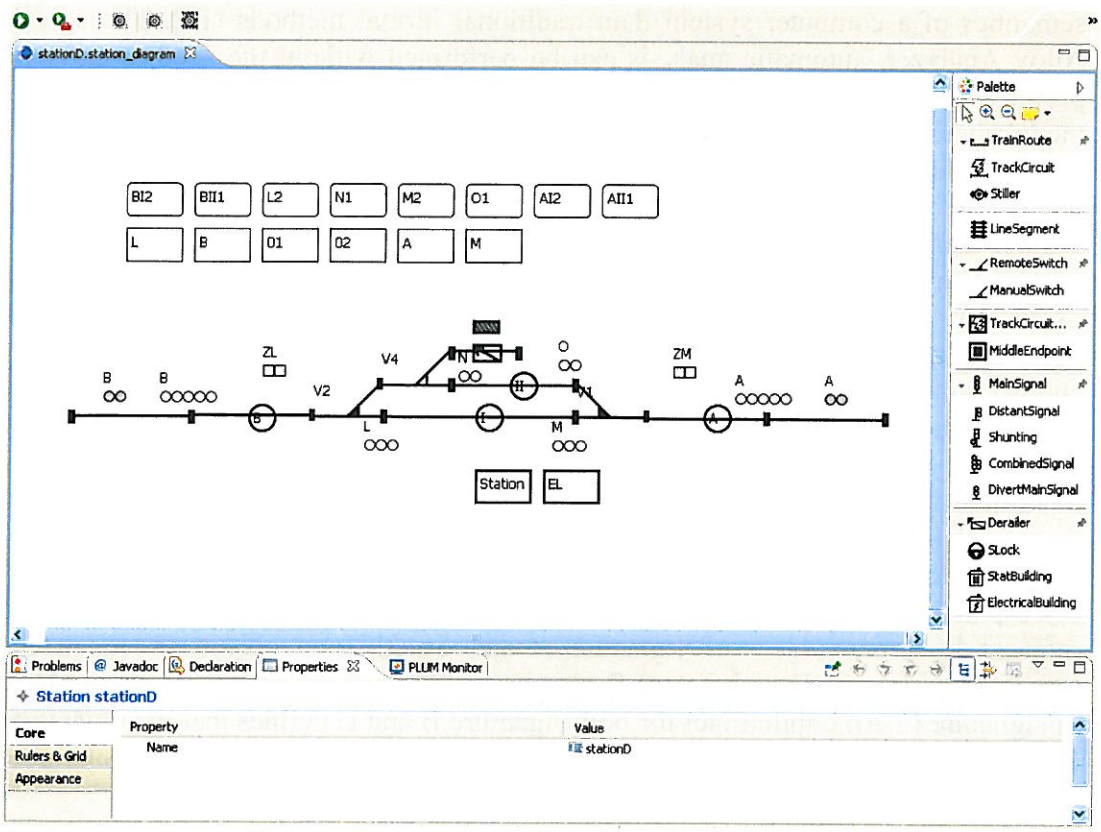

Fig. 5. TCL graphical editor

Initially, the TCL metamodel does not contain any constraints other than the association multiplicities in the metamodel. Thus the metamodel allows unintended models. Since TCL describes a safety-critical domain, it is vital that only syntactically and static semantically correct models are permitted.

2.3 Alloy

Traditionally, performing formal verification and validation of a computer system has required the system to be precisely expressed in mathematical terms to be able to prove the correctness of the system. There are languages and tool support (theorem provers) that perform such analysis of computer systems [13]. However, complex proofs cannot be fully automated, and they require assistance from an experienced user. The required knowledge of mathematical techniques with their complex notation, such as theorem proving, thus raises the threshold for performing analysis of computer systems.

Alloy is a light-weight declarative language, based on first-order logic, which offers automatic and incremental analysis through relational calculus [12]. It has been shown that Alloy has simpler and more uniform notation to define the formal semantics of a computer system than traditional formal methods [16]. Through the Alloy Analyzer, automatic analysis can be performed without the need to carry out proofs with complex mathematical notation. Unlike traditional theorem proving and model checkers, the Alloy Analyzer only guarantees the correctness and completeness of the result of the analysis up to a user-specified scope. This scope defines the number of instantiated objects of each type. However, the small scope hypothesis ensures that if a solution exists, it will be within a scope of small size [3].

An Alloy model typically consists of *signatures*, *fields*, *facts*, *predicates* and *assertions*. A signature defines a type in the model, which can be instantiated into objects. A type hierarchy of signatures can be defined by having a signature extending another signature. A signature contains fields, which refer other signatures, such that the instantiated objects can be related. Furthermore, a fact consists of a set of global constraints that must always hold. A predicate consists of a set of constraints that must hold if the predicate is processed. Finally, an assertion consists of constraints that are claimed to hold.

The Alloy syntax is illustrated in Fig. 6, which defines a small Alloy model corresponding to the metamodel in Fig. 2a. Signature *A* has two fields, *b* and *c*, which refer at least one object of signature *B* and exactly one object of signature *C* respectively. In addition, signature B also has a field referring to zero or one objects of signature C. An implicit fact for both signature B and C defines that the object of B and C has to be referred by exactly one object of A. This fact defines the containment relation.

Alloy offers two kinds of analysis: Find a solution to a predicate or find a counter-example to an assertion. Processing a predicate involves populating the signatures with objects up to the defined scope, to find a solution where all the constraints in the predicate and the global facts are satisfied. Alloy only guarantees that the solution satisfies all the constraints, and does not guarantee that this is the optimal solution or

that it holds any other kind of property. Similarly, finding a counter-example to an assertion involves populating the signatures with objects up to the defined scope, to find a model where the constraints in the assertion are not satisfied. The Alloy Analyzer performs analysis by translating the Alloy model to first-order logic which is used as input to a SAT solver. The result of the analysis is then translated back to an Alloy solution model.

```
sig A {
      b: some B,
      c: one C
}

sig B {
      c: lone C
}{ one a:A | this in a.b}

sig C {
}{ one a:A | this in a.c}
```

Fig. 6. Example model in Alloy

2.4 Object Constraint Language

OCL is a formal declarative language for defining rules and constraints for models in the Meta-Object Facility (MOF) [17]. OCL was originally developed for the Unified Modeling Language (UML), which is defined by MOF. However, Ecore is the Eclipse version of MOF, and consequently there are OCL implementations that support Ecore. Thus, OCL can define constraints directly on Ecore metamodels, to restrict the number of legal models according to the metamodel.

An OCL constraint is always defined within a context, such as a class. Since the purpose of OCL originally was to constrain UML models, it can be used for several purposes, e.g. specifying invariants on classes, pre- and post-conditions on operations and constraints on operations. Since we concentrate on Ecore metamodels without operations, we will apply invariants to the metaclasses.

Fig. 7 illustrates the concrete syntax of two invariants in OCL on the metamodel in Fig. 2a. Both invariants specify class A as context. The first restricts the number of contained B objects to be less than three, while the second specifies that for all contained B objects, their referred C object has to be equal to the contained C object.

```
context A
inv: self.b->size() < 3

context A
inv: self.b->forAll(b:B | b.c = self.c)
```

Fig. 7. OCL example

3 The Approach

To describe our approach, we first give a brief overview before we thoroughly describe the three steps of the approach applied on TCL. The three steps include generating models according to a metamodel, searching for unintended structures in the models, and enhancing and constraining the metamodel.

3.1 Overview of the Approach

Our starting point is a metamodel for a particular DSL. The metamodel describes the abstract syntax of the DSL, by capturing the concepts of the given domain and how the concepts are related. The metamodel defines the set of legal models in the DSL, in terms of model element object structures (that is objects and links between these). Furthermore, the metamodel can be enriched with logical constraints which can prohibit undesired structures. Constraints that describe the dynamic behavior of the language can also be added (see Section 5).

The approach involves three steps: Step 1 of the approach generates a formal definition (in Alloy) of the static semantics, based upon the metamodel, including related constraints. We will name this definition the *Alloy model*. The Alloy Analyzer is then used to find a model satisfying the generated semantics, namely a *solution model*. This solution model is transformed back into a model in the DSL, which can be viewed by the DSL editor. Note that all parts of step 1 are performed automatically. Step 2 involves using the generated DSL model to search for unintended structures in the model. This is a manual step which requires domain knowledge to distinguish the models that are unintended. Step 3 involves using this knowledge to recognize how further enhancements and constraints can be applied to the metamodel. Adding the constraints and enhancing the metamodel is performed manually based on the knowledge from step 2. However, the added constraints can be transformed automatically between Alloy and the metamodel constraint language. The whole process, including the three steps, is depicted in Fig. 8. Note that step 1 and 3 are divided into sub-step 1.1, 1.2 and 1.3, and 3.1 and 3.2 respectively.

3.2 Generating Models

There are several means for finding weaknesses of a metamodel, e.g. analyzing the metamodel to prove certain properties of it, and creating a number of models according to the metamodel to reveal the weaknesses. Our approach focuses on the latter technique. By studying a set of models the modeler can decide whether they comply with the intention of the metamodel.

From Metamodel to Alloy Model

Metamodels represent concepts of a domain by classes, their attributes and associations. Alloy can express complex structural constraints. These constraints can be used to represent similar structures as the metamodel, such as using signatures to

express the concepts, fields to express associations, facts to express constraints etc. We have summarized the mapping between the elements in a metamodel and an Alloy model in Fig. 9. This is based on the work by Kelsen and Ma and their comparison of traditional formal methods and Alloy [16].

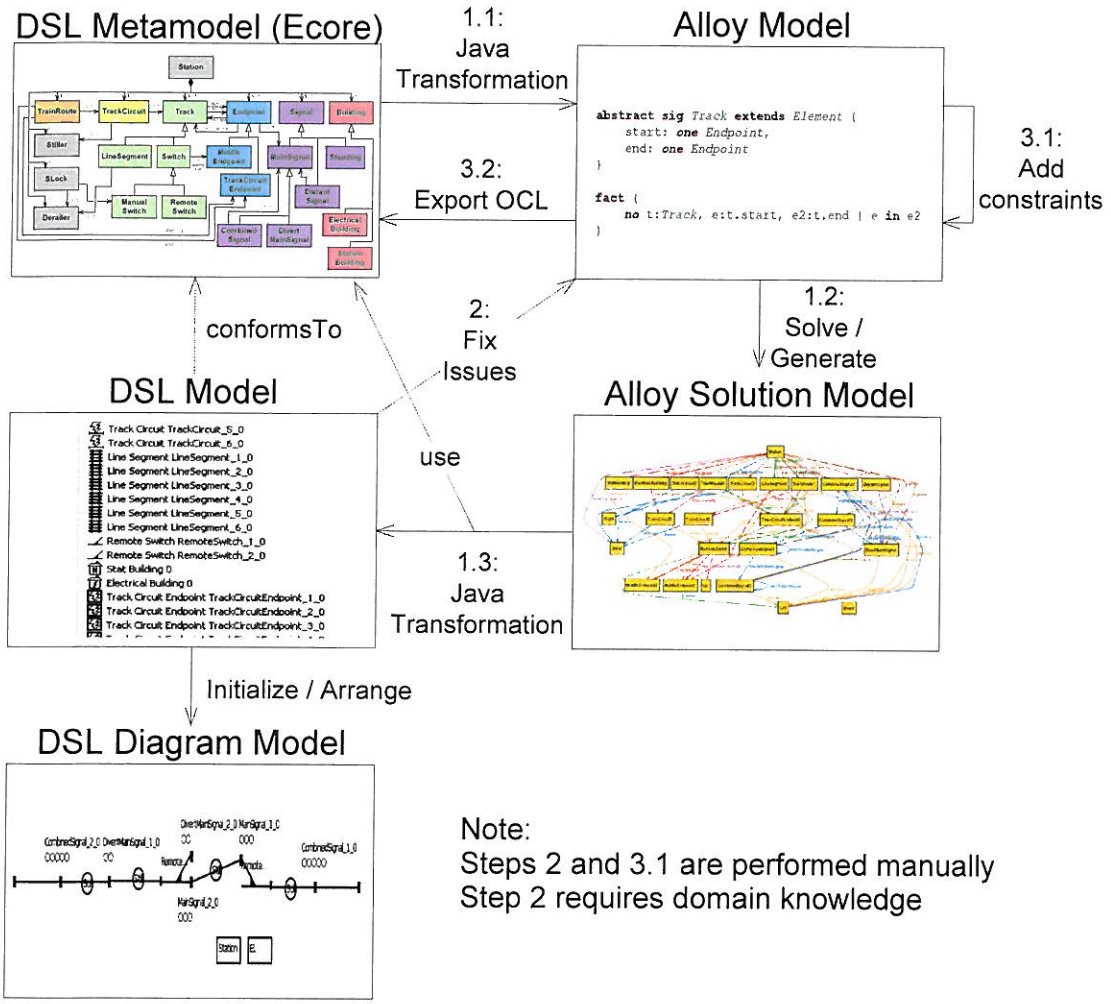

Fig. 8. Processes and artifacts of the approach

The transformation to an Alloy model is generic and transforms each class of the metamodel to a signature, each association to a field within the corresponding signature, and the association multiplicities to the corresponding Alloy multiplicities (*lone*, *some* etc.). To illustrate the generation of an Alloy model, we have applied this generator to the initial TCL metamodel (see Fig. 10). The hierarchy of the Alloy model follows the hierarchy of the metamodel, with *Station* as the root element. For each signature, we add a constraint for correctly representing containment associations.

Ecore Metamodel	Alloy Model
Class	Signature
Association	Field
Enum E(A, B, ...)	Signature E One Signature A extends E One Signature B extends E
Attribute	---
Constraint (OCL)	Fact

Fig. 9. Mapping between Ecore metamodel and Alloy model

```
one sig Station {trainRoutes: some TrainRoute, trackCircuitsInStation: some TrackCircuit,
      tracksInStation: some Track, buildings: set Building, endpoints: some Endpoint,
      signals: some Signal, stillersInStation: set Stiller, sLocksInStation: set SLock,
      derailersInStation: set Derailer}
sig TrainRoute {trackCircuits: some TrackCircuit, start: one TrackCircuitEndpoint,
      end: one TrackCircuitEndpoint, direction: one Direction}
      {one st:Station | this in st.trainRoutes}
sig TrackCircuit {stillers: lone Stiller, tracks: some Track}
      {one st:Station | this in st.trackCircuitsInStation}
sig Stiller {} {one st:Station | this in st.stillersInStation}
abstract sig Building {} {one st:Station | this in st.buildings}
sig StatBuilding extends Building {}
sig ElectricalBuilding extends Building {}
abstract sig Track {start: one Endpoint, end: one Endpoint}
      {one st:Station | this in st.tracksInStation}
sig LineSegment extends Track {derailer: lone Derailer}
abstract sig Switch extends Track {oneSidePort: one MiddleEndpoint,
      direction: one Direction, diversionDirection: one DiversionDirection}
abstract sig Endpoint {connectedMainSignal: lone MainSignal, first: lone Track,
      second: lone Track} {one st:Station | this in st.endpoints}
sig TrackCircuitEndpoint extends Endpoint {}
sig MiddleEndpoint extends Endpoint {}
sig RemoteSwitch extends Switch {}
sig ManualSwitch extends Switch {}
abstract sig Signal {direction: one Direction} {one st:Station | this in st.signals}
sig MainSignal extends Signal {}
sig DistantSignal extends Signal {maindistantSignal: some MainSignal}
sig Shunting extends Signal {}
sig CombinedSignal extends MainSignal {mainDistantSignal: some MainSignal}
sig DivertMainSignal extends MainSignal {}
sig Derailer {} {one st:Station | this in st.derailersInStation}
sig SLock {lockDerailer: lone Derailer, lockSwitch: one ManualSwitch}
      {one st:Station | this in st.sLocksInStation}
abstract sig DataType {}
abstract sig Direction extends DataType {}
abstract sig DiversionDirection extends DataType {}
one sig Left extends Direction {}
one sig Right extends Direction {}
one sig Up extends DiversionDirection {}
one sig Down extends DiversionDirection {}

pred show{}
run show for 20
```

Fig. 10. Initial Alloy model of TCL

Constraints may already be applied to the metamodel prior to the initial generation of the Alloy model. OCL is a widely used constraint language for both UML and

Ecore. These constraints may restrict the number of possible models according to the metamodel, and should therefore also be translated to Alloy facts. Anastasakis et al. [1] present a correspondence table between OCL and Alloy in their work on analyzing UML models using Alloy. We use a subset of this correspondence table between OCL and Alloy to make a small parser that parses OCL syntax and generates Alloy constraints. Consequently, the restrictions already defined on the metamodel can be transformed to also restrict the generated Alloy model. It is worth noticing that this transformation is quite restricted and does not support full OCL.

From Alloy Model to DSL Model

Using the generated Alloy model as input, we invoke the Alloy Analyzer to produce an arbitrary solution model satisfying the structure and constraints in the Alloy model. Fig. 11 illustrates an arbitrary generated solution model from the Alloy model in Fig. 10. Note that the object diagram in the figure is used for illustration purposes, and the details of it are not significant. The object diagram can be used to search for unintended structures. However, it is a tedious process to study all the objects and follow all the links to check the correctness of the model. Representing the model in the syntax of the DSL itself is therefore important, and we consequently transform the solution model into a DSL model according to the original metamodel.

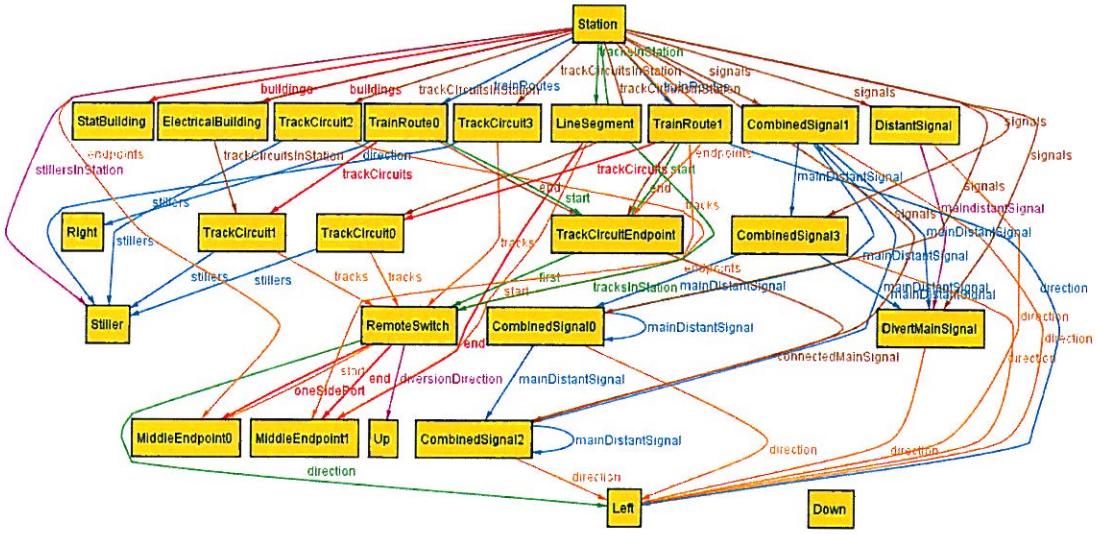

Fig. 11. Alloy solution model for TCL

To perform this transformation we require the user to import the metamodel to extract information about all the necessary classes. This information is used to instantiate the appropriate classes using dynamic EMF [24]. Dynamic EMF allows an Ecore-based model to be built at runtime, without the need for implementation classes from the metamodel. The transformation is divided into two steps: First the objects are instantiated, and then the associations between the objects are specified properly.

Transforming an Alloy solution model for TCL yields a TCL model according to the TCL metamodel. This TCL model only describes the model elements and their properties. Since TCL is a language with a graphical syntax, the result ought to be

shown using the graphical syntax. GMF provides the opportunity to initialize the diagram based on the model, however, without specified position of the elements. The TCL tool therefore includes functionality to arrange stations by calculating the graphical positions of the elements on the diagram. The initial solution station yields the arrangement result shown in Fig. 12.

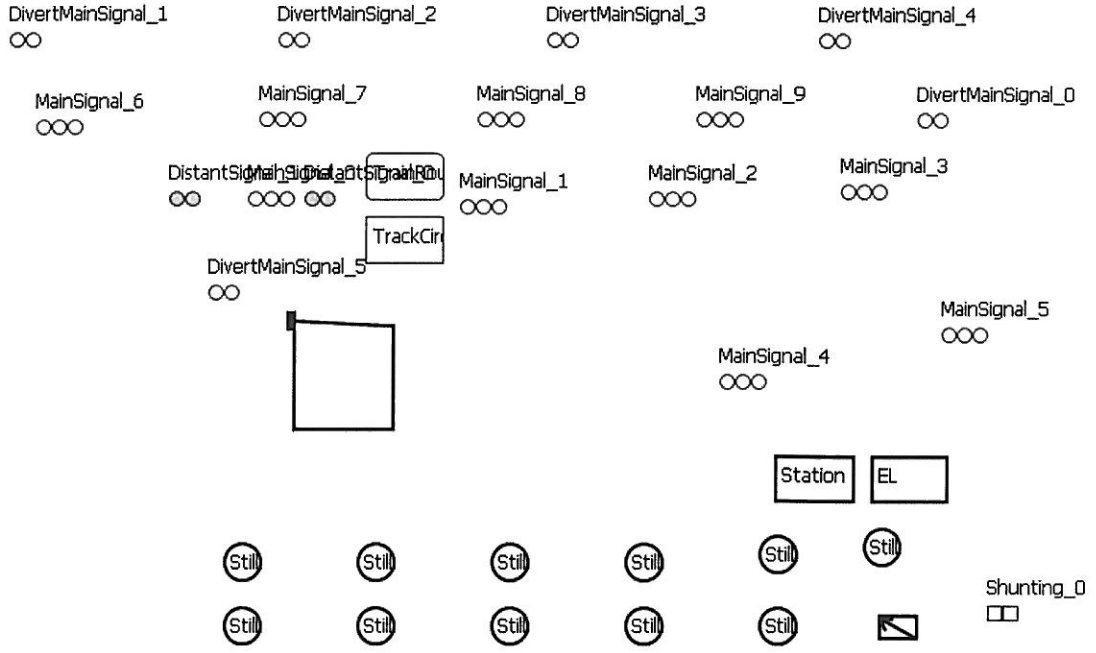

Fig. 12. Initial TCL model

As expected, the figure demonstrates that we are able to use TCL to model clearly irrational and unintended stations. This model basically contains a random set of objects within the scope, with a random set of associations satisfying the metamodel. The figure illustrates that there is a need for restricting how the model elements are related to each other.

3.3 Finding the Issues

As illustrated in Fig. 12, the initial TCL metamodel (without constraints) allows unintended TCL models to be created. By a careful examination of the generated model, we can find the issues and constrain the metamodel to exclude them. We give four steps on how to discover the issues leading to the unintended models. Note that these four steps require domain knowledge to distinguish unintended structures in the generated models.

The first step is to check the basic connectivity between the elements in the model, to see if the model element objects are related as expected. The question is whether all objects are referred and contained as expected for an intended model. For instance, a model may have loops or indirect loops, even though this is not intended by the DSL designer. E.g. TCL is not supposed to contain loops or indirect loops (other than stations with more than one track, which are connected by switches). However, as

illustrated in Fig. 12, a LineSegment is referring the same Endpoint for the *start* and *end* association. Furthermore, a TrainRoute is always supposed to start at a TrackCircuitEndpoint with a single connected MainSignal. Since the generated model contains more than 10 MainSignals and only 1 TrackCircuitEndpoint, we can clearly see that there are MainSignals without this connection.

The second step considers the meaning of the relations between objects in the model beyond the relations between single objects. We search for semantic duplication of objects, whether the references of referred objects are as intended and whether values of data types are as expected. Semantic duplication of objects involves having two or more objects referring the same set of objects. E.g. in TCL a TrainRoute is supposed to start at one TrackCircuitEndpoint and end at another TrackCircuitEndpoint. However, the TCL metamodel allows two TrainRoutes to start and end at the same places, which is not intended. Furthermore, it is necessary to confirm that the values of data-types are as intended. E.g. switches in TCL have a direction, which can be *LEFT* or *RIGHT*.

The third step involves looking for the number of objects relative to each other. As illustrated in Fig. 12, there are only one TrainRoute, but more than 10 MainSignals. Since a TrainRoute starts at a TrackCircuitEndpoint with a connected MainSignal, the number of MainSignals is restricted to be equal or less than the number of TrainRoutes and the number of TrackCircuitEndpoints. It is worth noticing that by constraining the connectivity of the objects in the previous steps, some of the number issues may be implicitly solved.

The fourth step involves an overall view of a set of the generated models to see if the metamodel is not restrictive enough, if it is too restrictive and whether it performs as intended. We add restrictions that go beyond the constraints on single objects and their references. An example in TCL is as follows: For each TrainRoute into a station there must be a subsequent TrainRoute out of the station in the same direction. However, these TrainRoutes are not directly related, and thus require constraints that take the overall structure of the station into account.

To figure out whether the metamodel is too restrictive or not restrictive enough, we use assertions to check certain properties. This allows us to check if models exist with the asserted structures. If the metamodel allows such a structure, the Alloy Analyzer returns a counter-example in the form of a model with this structure, which is transformed to a DSL model.

Fig. 13 summarizes the steps in the approach. It begins with an initial DSL model, suggests what to look for in each step, and ends with a final DSL model. For each revealed issue, constraints should be added and the DSL model regenerated. The intermediate models illustrated in the figure (e.g. *Second model*) are a fraction of all the intermediate models generated during the procedure.

3.4 Enhancing and Constraining the Metamodel

Based on the issues revealed by following the procedure described in Section 3.3, we enhance and constrain the metamodel incrementally. The result of the first part of the approach is an Alloy model, which is used to generate solution models that are transformed back to DSL models. Based on inspections of these DSL models

performed by domain experts, issues with the metamodel are revealed. To solve these issues, we add constraints to the Alloy model. We then generate a new solution model that is transformed back to a DSL model. This procedure continues until we have walked through the process of revealing issues, and are thus satisfied with the generated models. Then we transform the constraints, represented in Alloy, to constraints in the metamodel constraint language. In this approach we have used Ecore as a base, and thus we provide a translation from Alloy constraints to OCL constraints. As we will see in Section 4, DSL models can be validated to ensure that they satisfy the inserted constraints.

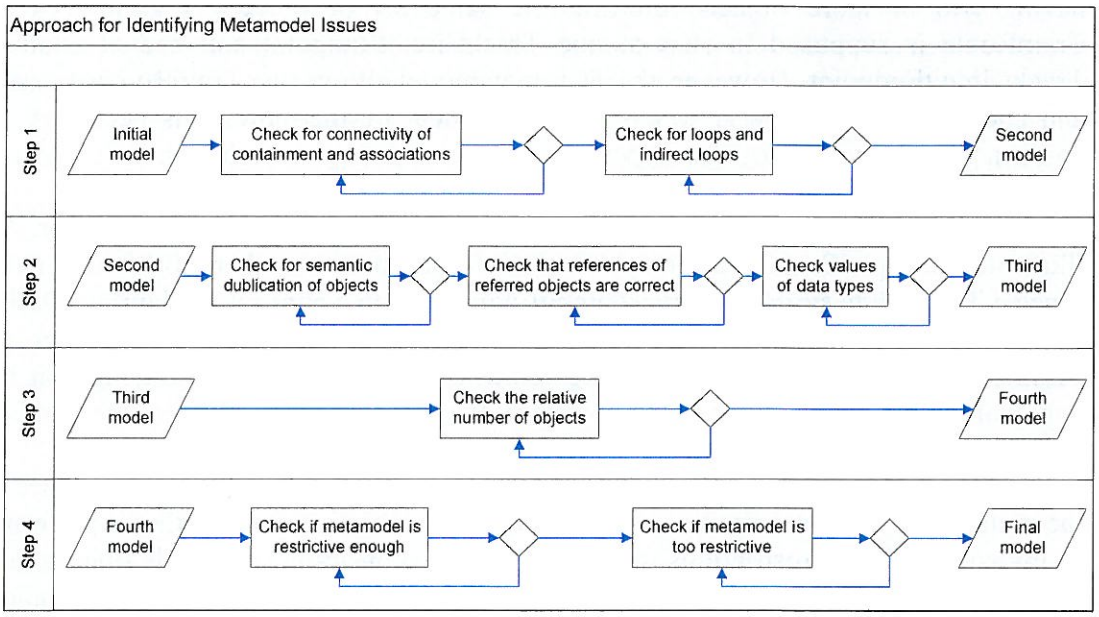

Fig. 13. Approach for identifying metamodel issues

Adding constraints to a DSL is not trivial, and this approach requires that the DSL developer is either familiar with Alloy or the metamodel constraint language (e.g. OCL). With Alloy knowledge the constraints can be directly added to the Alloy model, and later transformed to the metamodel constraint language. With only knowledge of the metamodel constraint language, specified constraints have to be transformed to Alloy before the generation of solution models. Whether to use Alloy or OCL to define the constraints is dependent on the preference of the developer.

We follow the procedure from Section 3.3 to add constraints to the initial Alloy model. Fig. 14 shows a list of the constraints we have added to TCL. We will only show the implementation and result of adding some of them.

The first step in the procedure is to define constraints to improve the connectivity of the elements in Fig. 12. We show two constraints that have been added to improve how the objects are related (see Fig. 15). The first Alloy constraint ensures that all MainSignals are placed at the start of a TrainRoute. In other words, for all MainSignals, there exist at least one TrainRoute which is connected to this MainSignal through its start TrackCircuitEndpoint. The second constraint guarantees that we do not have any direct loops of Tracks. In other words, no Track exists such that *start* and *end* refer the same objects.

```
Step 1:
A Stiller can only be referred by one TrackCircuit.
A Track can only be referred by one TrackCircuit.
A MainSignal can only be referred by one TrackCircuitEndpoint.
A ManuelSwitch can only be referred by one SLock.
A Derailer can only be referred by one LineSegment.
A Derailer can only be referred by one SLock.
TrainRoute start/end should refer different Endpoints.
Correspondance between the references between Endpoint and LineSegment.
Correspondance between the references between Endpoint and Switch.
Track start/end should refer different Endpoints.
Endpoint first/second should refer different Tracks.
DistantSignal and referred MainSignal have the same direction.
CombinedSignal and referred MainSignal have the same direction.
CombinedSignal should not refer itself.
All MainSignals are referred from a TrainRoute start.
MiddleEndpoint should not refer a MainSignal.

Step 2:
Specify the left side of the station.
Specify the right side of the station.
TrackCircuitEndpoints should divide a TrackCircuit.
MiddleEndpoints should not divide a TrackCircuit.
Only one TrainRoute can start and end at a given place.
All TrackCircuits must be connected in a TrainRoute.
The TrackCircuits after TrainRoute start and before TrainRoute end
     are in the TrainRoute.
Correspondance between the direction of Switch pairs.

Step 3:
The number of TrainRoutes is restricted to 4, and 4 more per
     RemoteSwitch pair.
Maximum two buildings per Station.

Step 4:
An ingoing TrainRoute is ending at the same place as another outgoing
     TrainRoute is starting.
MiddleEndpoint should have a LineSegment in one end.
```

Fig. 14. TCL constraints

After adding a few of the constraints in step 1 in Fig. 14 we generate a new solution model which is transformed into a DSL model. From this DSL model the diagram is initialized and the station arranged. The result is illustrated in Fig. 16, where we can see that there are e.g. no loops and all MainSignals are referred by a TrainRoute through a TrackCircuitEndpoint. However, there are still some challenges with the connectivity and the relative number of objects. For instance, there are several Endpoints without any Tracks between them, and the number of TrainRoutes, DistantSignal (signals at the top left) and Endpoints are relatively high. Furthermore, when arranging the station in TCL, we get a message telling us that the algorithm could not find the left or right ends of the station.

```
fact {
    all ms:MainSignal {
        some tr:TrainRoute | tr.start.connectedMainSignal = ms
    }
}

fact {
    no t:Track, e:t.start, e2:t.end | e = e2
}
```

Fig. 15. All MainSignals starts a TrainRoute and no direct loops of Tracks

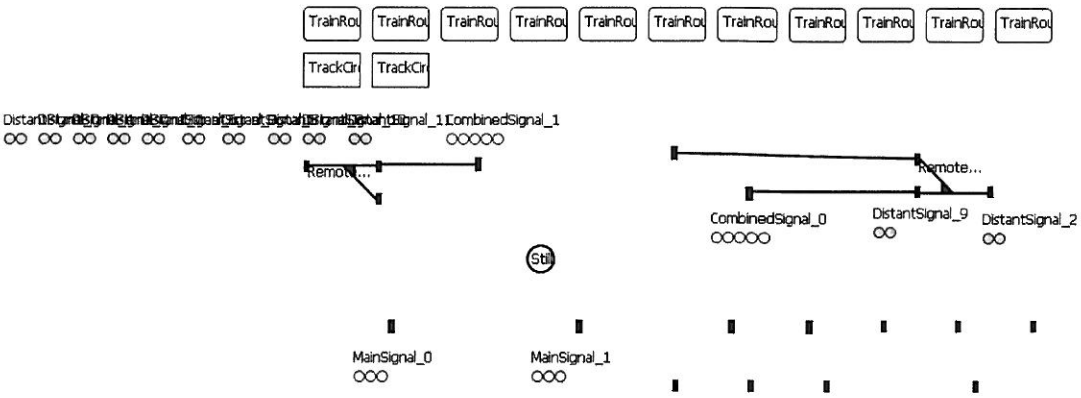

Fig. 16. Second version of the generated TCL model

Even though it may be quite clear for a modeler that an Endpoint to the left on the diagram is the left of the station, the model does not have a precise definition of the left or right ends of the station. Therefore, there is no general possibility for expressing how the left and right side of the station should be by using constraints. Consequently, we have extended the TCL metamodel to also define the left and right sides of the station by adding two associations, *left* and *right*, from Station to TrackCircuitEndpoint.

After adding the rest of the constraints in the first step, in addition to the left and right associations (step 2), we end up with the station illustrated in Fig. 17. We can identify the left and right of the station and the objects seems to be related. However, there are only three TrackCircuits (squared boxes), while there are five Tracks that are divided by TrackCircuitEndpoints. We therefore continue with the next step, to identify the meaning of the relations between objects in the model. Note that the model illustrated in the figure is smaller than the previous generated model, because Alloy generates arbitrary models.

Continuing further on the second step, we add constraints for correctly dividing the Tracks into TrackCircuits (see Fig. 18). This constraint uses TrackCircuitEndpoints to divide TrackCircuits. In other words, for all TrackCircuitEndpoints except the left and right side of the station, there exist one TrackCircuit *tc* and one TrackCircuit *tc2*, such that the Track to the left of the TrackCircuitEndpoint is related to *tc*, and the Track to the right of the TrackCircuitEndpoint is related to *tc2*.

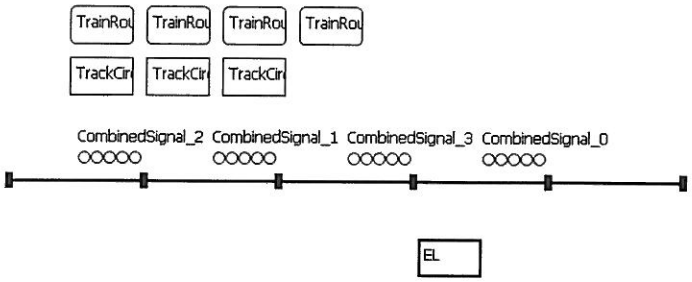

Fig. 17. Third version of the generated TCL model

```
fact {
    all st:Station, tce:TrackCircuitEndpoint - st.left - st.right {
        one tc, tc2:TrackCircuit {
            tce.first in tc.tracks
            tce.second in tc2.tracks
        }
    }
}
```

Fig. 18. Divide TrackCircuits correctly

Adding the other constraints of the second step, and also implementing the constraints in the third step, yields the model illustrated in Fig. 19. As we can see, the number of objects and the structure now seems to be more as expected. There is, however, a gap between the two Switches. Furthermore, the MainSignals are not correctly placed, as can be seen from the left side of the station where two MainSignals are placed after each other. Thus, the TrainRoutes are not starting at the right places. We therefore follow the procedure in the fourth step and add a couple of more constraints to properly connect ingoing and outgoing TrainRoutes, such that an outgoing TrainRoute starts where an ingoing TrainRoute ends. The result of adding these constraints is illustrated in Fig. 20, which finally seems to be a reasonable station model.

When the fourth step, with several iterations, is finished, the constraints can be transformed and appended to the metamodel. We use a subset of the correspondence table from [1] to define a small parser, which parses Alloy syntax and generates OCL constraints. Note that we only use parts of the correspondence table, since one statement in Alloy can be translated to more than one kind of statement in OCL (e.g. Alloy statements with the *in* construction). Also notice that we have extended the table with two statements to support the Alloy quantifiers *one* and *lone* (see Fig. 21).

The transformed OCL version of the first constraint in Fig. 15 and the constraint in Fig. 18 is illustrated in Fig. 22. In Section 4 we show how these constraints can be added to an Ecore metamodel to constrain the models accordingly.

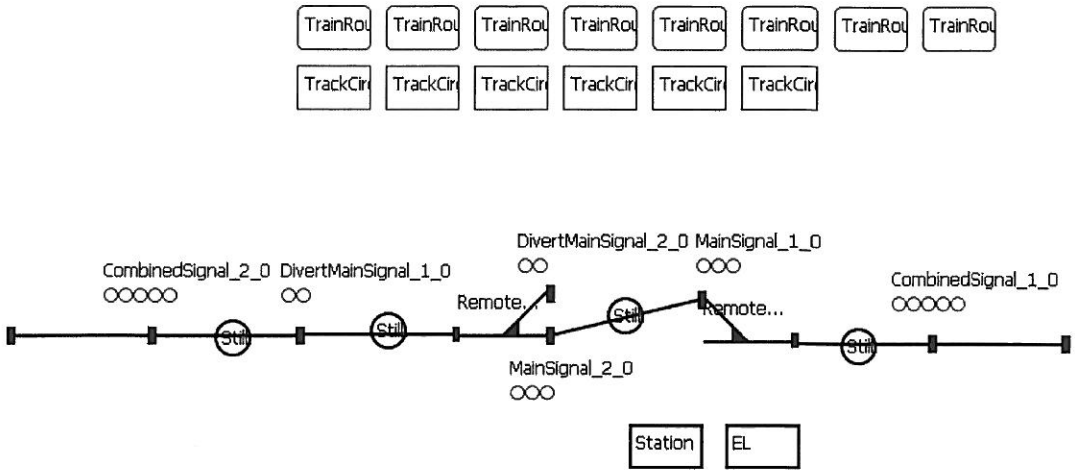

Fig. 19. Fourth version of the generated TCL model

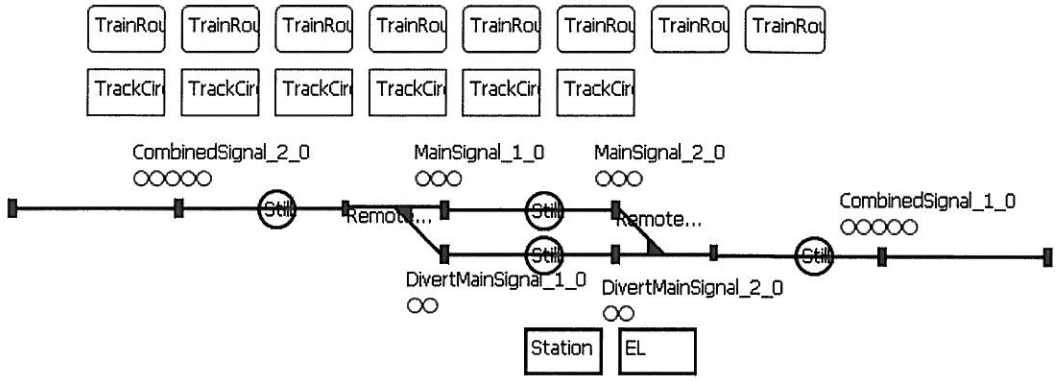

Fig. 20. Final version of the generated TCL model

OCL Expression	Alloy Expression
col -> exists(obj | exp)->size() = 1	one obj : col | exp
col -> exists(obj | exp)->size() <= 1	lone obj : col | exp

Fig. 21. Extra mapping between OCL and Alloy

```
context Station
inv: self.signals->select(s | s.oclIsKindOf(MainSignal))-> forAll(ms |
self.trainRoutes->exists(tr | tr.start.connectedMainSignal = ms))

context Station
inv: self.endpoints->select(ep | ep.oclIsTypeOf(TrackCircuitEndpoint))->
excluding(self.left)-> excluding(self.right)->forAll(tce |
self.trackCircuitsInStation-> exists(tc | self.trackCircuitsInStation->
excluding(tc)-> exists(tc2 | tc.tracks->includes(tce.first) and tc2.tracks->
includes(tce.second))->size() = 1)->size() = 1)
```

Fig. 22. OCL constraints

4 Implementation

The approach for improving metamodels is generic, however, the discussion and implementation in this paper is based on the Eclipse platform and Ecore. We have implemented a prototype to support the approach as an Eclipse plug-in. This plug-in supports generic generation of Alloy models from Ecore metamodels (with OCL constraints), generic generation of DSL models from an Alloy model and transformation of Alloy constraints to OCL. This prototype plug-in therefore supports the procedure defined in Section 3.

The generation of an Alloy model from an Ecore metamodel is implemented in Java using the EMF Framework. By traversing all classes, associations and data-types, we generate equivalent Alloy signatures, fields and implicit facts. In addition, the OCL constraints in the metamodel are parsed and translated to Alloy. This is, however, a restricted parser to show the feasibility of this transformation, and does not consider the full OCL language. The plug-in extends the *popupMenus* extension point to extend the Eclipse user interface with a new menu item for initiating the generation of Alloy models. Fig. 23 illustrates this user interface integration when selecting an ecore-metamodel. The result of this generation is a new file containing the generated Alloy model.

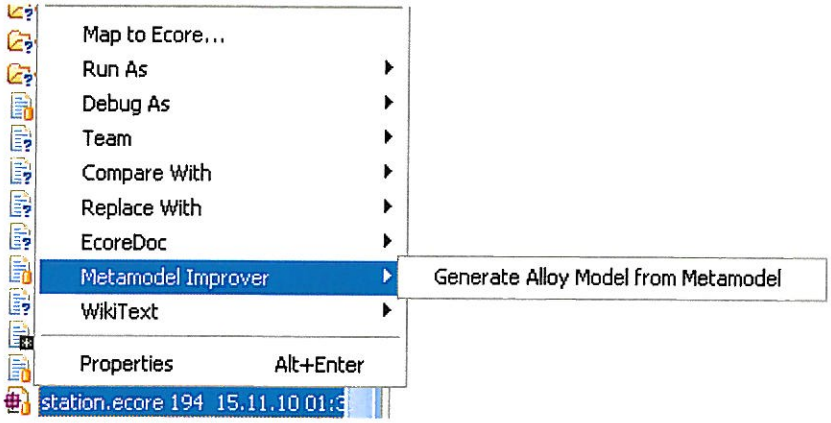

Fig. 23. Interface for generating Alloy model

The Alloy model can be enhanced with constraints according to the procedure discussed in Section 3. Furthermore, the tool allows the user to select an Alloy model for generating a DSL model (see Fig. 24). The Alloy Analyzer is then invoked and the solution model is traversed to extract all the objects and how they are related. The tool uses dynamic EMF [24] to instantiate the proper objects based on the Alloy model and the Ecore metamodel. Note that this generates a repository model, and if the DSL has a graphical syntax, the diagram model has to be initialized using the DSL tools. TCL provides tool support for initializing the diagram model and arranging stations.

Using the user interface provided by the tool, OCL constraints can be exported from an Alloy model. The result of exporting the OCL constraints is a file with the constraints listed. These constraints can then be appended to the context element in the Ecore metamodel using the OCL for Ecore plug-in [20]. Regenerating the model

code for the editor will then yield validation code for automatic validation of DSL models. E.g. TCL models can be validated, where a violation of a constraint will yield an error message (see Fig. 25).

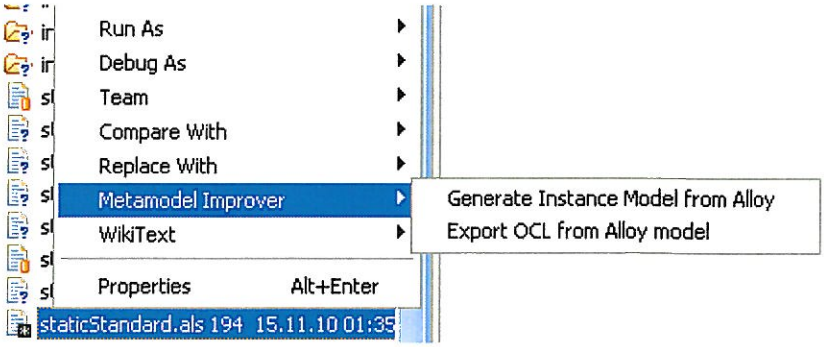

Fig. 24. Interface for generating DSL models and OCL constraints

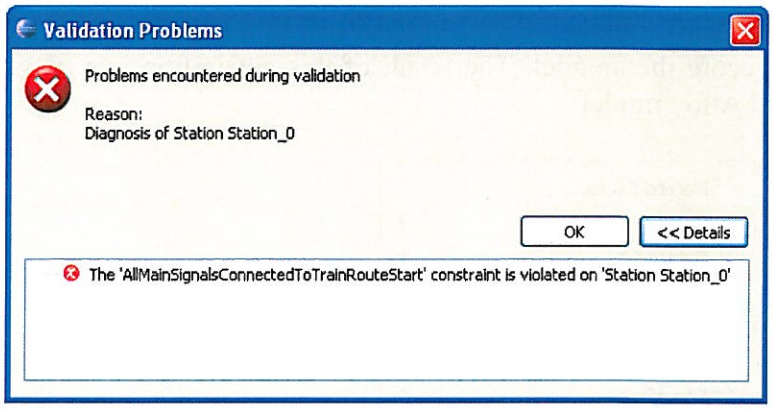

Fig. 25. Violation of an OCL constraint in an EMF editor

5 Discussion

The approach described in this paper relies on the generation of arbitrary models to uncover unintended characteristics of a metamodel. A DSL designer would typically design the metamodel with the most obvious constraints. This approach is then useful to reveal further restrictions needed on the metamodel. Even though the approach does not guarantee that all issues will be solved, the generation of models is generic and automatic, and thus several models can be generated to maximize the probability of finding important constraints. Furthermore, since the models are arbitrarily created, there is no user-knowledge to guide the construction of the model, leading to a higher possibility of generating an unintended model. Our experience shows that during the procedure of improving the TCL metamodel, a great majority of the generated models are not as intended, even after adding a large amount of constraints. The approach can

therefore assist the developer to improve the metamodel, but will not guarantee an optimal result.

Another concern is that specifying constraints in Alloy, or in another constraint language, is not trivial. However, the need for expressing constraints on languages is inevitable, and whether to use Alloy or the metamodel constraint language (e.g. OCL), depends on the preference of the developer. We provide a transformation between a restricted set of Alloy and a restricted set of OCL to avoid forcing the use of one of them. The approach allows the developer to generate models automatically, such that the developer can experiment with the effect of different kinds of constraints. This can increase the confidence in the added constraints.

The Alloy Analyzer will find and return a solution model satisfying all the constraints in the model. It is, however, possible to add constraints that are conflicting, such that no legal model can be instantiated. The approach described in this paper encourages incremental constraining and enhancement of the metamodel, meaning that models should be generated for each added constraint. Modifying the most recent added constraints can therefore solve this issue. Another concern is when over-constraining leads to a subset of the intended models being prohibited. In this case, we suggest the use of assertions to confirm that some characteristics of the metamodel still are permitted.

Alloy is based on first-order logic and the Alloy Analyzer will search for all possible solutions within the user-specified scope. When the DSL models grow in size, this scope becomes larger and the number of possibilities grows exponentially. Optimizing the Alloy model by reducing the complexity of the expressions and using partial instances, which involves using constant functions to resolve associations, can reduce the analysis time needed. However, such optimizations should be used with care, since they can implicitly restrict the associations in the model.

In this paper we have applied the approach to TCL, which has a graphical nature. The approach has proved to be useful for this DSL, and we expect that it has similar effect on other graphical and textual DSLs. However, case-studies using the approach in other DSLs in other domains are necessary for confirming this expectation. Yet, we do not see big differences between TCL and other DSLs that will significantly affect the usefulness of this approach.

We have so far concentrated on the use of the static semantics of the DSL. Svendsen et al. [26] propose an approach for performing simulations and dynamic analysis of TCL models. They propose a formalization of the static and dynamic semantics of TCL in Alloy. The dynamic semantics defines a state-machine by defining a set of states and how the properties of a station change in the transition between each state. The Alloy Analyzer is used to find a trace through the state-machine for a particular behavior of the station.

We see an added value of using such dynamic semantics to further validate the generated solution models and thus the metamodel. The generated solution models can then be simulated or analyzed using the dynamic semantics. Other issues with the metamodel can be discovered than for the pure static analysis. However, this requires the developer to use Alloy to model the dynamic semantics of the DSL, which is not a trivial task. Our experience from the TCL example shows that this can be a useful fifth step of the approach.

6 Related Work

Herrmannsdoerfer et al. [11] present an approach for improving metamodels by analyzing a large set of models to gather information about the usage of the metamodel concepts. Based on the metamodel expectations, defined automatically and manually, the information is used to suggest metamodel improvement. While they use statistics gathered from a set of models, based on user experience, to suggest improvements, our approach generates models automatically to reveal necessary improvements. Thus, our approach does not rely on a large set of models being manually created.

Gogolla et al. [10] present an approach for generating snapshots (an instance model at a given position in a trace) of UML models. They extend the UML Specification Environment (USE) with the language ASSL (A Snapshot Sequence Language) for automating the generation of snapshots based on some specified properties. They use the snapshots to test the UML models, with corresponding OCL. Their approach requires the user to specify procedures for how to generate the objects and links using ASSL. Our approach generates models automatically, and is not limited to UML.

Sen et al. [22] propose an approach for extracting a required subset of a large metamodel. Based on a large metamodel and a set of required classes and properties, their algorithm finds all mandatory dependencies between the required concepts, and generates an output metamodel with only these concepts. Our approach is more general, and is concerned with restricting the metamodel to avoid unintended models.

Instead of suggesting improvements to a particular metamodel, some works discuss guidelines to follow when developing metamodels. Kelly and Pohjonen [14] discuss common pitfalls of domain-specific modeling and guidelines to follow to avoid them. Kelly and Tolvanen [15] give a thorough introduction to domain-specific modeling. They discuss the fundamentals of domain-specific modeling and present several examples of how domain-specific modeling can be applied.

There are several works concerned with the generation of models according to a metamodel. Anastasakis et al. [1] transform UML models, with OCL constraints, to Alloy to perform automatic analysis on these models. They provide a mapping between UML and Alloy using a UML profile, and between OCL and Alloy, and a tool prototype. Shah et al. [23] extend the approach with a transformation of Alloy solution models to UML object models, which correspond to the original UML class diagram models. In comparison, our approach considers general metamodels, and offers a generic transformation between the metamodel and Alloy and back to the model corresponding to the metamodel. Furthermore, our approach does not focus on the analysis of UML models, but rather on metamodel improvement. Note that our implementation of the transformation between OCL and Alloy is based on the mapping presented in their work.

Ehrig et al. [5] introduce an approach for generating instance models using instance-generating graph grammars. They derive graph grammar rules from the metamodel, to create an instance of each class and associations between them. These rules are executed an arbitrarily number of times to get all possible models within a certain scope. Their work has been extended by Winkelmann et al. [29] to support a restricted set of OCL. Instead of formulating graph grammar rules, we transform the metamodel into an Alloy model, and execute the Alloy Analyzer to generate models.

This also gives opportunities to perform other kinds of analysis, such as checking for certain properties of the metamodel by defining assertions or dynamic simulations.

Sen et al. [21] present an approach for automatically generating models for black-box testing. They transform a metamodel to Alloy, generate solution models, both random and guided models, and transform these models back to the DSL for use as input in black-box testing. Their approach is similar and performing many of the same steps as our approach. However, our approach is concerned with improvements of metamodels, and thus also gives a procedure for how to identify unintended models and subsequently improve the metamodel.

Mougenot et al. [18] address the challenge of validating industrial tools and approaches. They suggest an approach, which is based on the Boltzmann method, for generating huge metamodel instances for this purpose. However, even though our approach also generates metamodel instances, it presents the instance model for inspection, such that flaws in the metamodel can be detected and improved.

There are several other works using Alloy for analysis of models and metamodels. Wegmann et al. [28] define a metamodel in Alloy to validate the models created by their tool. Van Der Straeten [25] uses Alloy to analyze UML models to find inconsistencies. Kelsen and Ma [16] present a comparison between traditional techniques and an Alloy approach for formalizing modeling languages. Georg et al. [8] give a comparison between the application of Alloy and OCL. Baresi and Spoletini [4] propose an approach for using Alloy to analyze graph transformation systems. Anastasakis et al. [2] describe an approach for using Alloy to analyze a model transformation and the well-formedness of the target model. Gheyi et al. [9] specify a theory for feature models in Alloy.

7 Conclusion and Future Work

This paper has presented an approach for improving DSLs by revealing unintended models, and subsequently improving the constraints and enhancements of the metamodels. We use Alloy to generate models corresponding to a metamodel, give a procedure for identifying unintentional models and show how to further constrain and enhance the metamodel. The approach has been demonstrated on an example DSL from the train domain, the Train Control Language. Furthermore, we walked through a prototype supporting our approach, before we discussed the approach and presented related work.

Performing case studies on other examples in other domains is important as future work to further evaluate the approach. We also see that an analysis of constraints may suggest metamodel improvements based on constraint patterns, which will be tried out in future work. For instance if a constraint restricts the use of an association, this association should probably be re-factored in the metamodel. Furthermore, automation of the search for unintended models, and a framework for simpler specification of constraints would be beneficial. Extension of the transformation between Alloy and OCL is also valuable future work.

Acknowledgements. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 and the Verde project ITEA 2 – ip8020 parts of the Eureka framework.

References

1. Anastasakis, K., Bordbar, B., Georg, G., and Ray, I., "On Challenges of Model Transformation from Uml to Alloy", in Software and Systems Modeling, vol. 9: Springer Berlin / Heidelberg, (2010), pp. 69-86
2. Anastasakis, K., Bordbar, B., and Küster, J.M., "Analysis of Model Transformations Via Alloy", in 4th International Workshop on Model Driven Engineering, Verification and Validation, B. Baudry, A. Faivre, S. Ghosh, and A. Pretschner, Eds. In conjunction with MoDELS07, Nashville, TN, USA: Springer, (2008)
3. Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D., "Evaluating the "Small Scope Hypothesis"," MIT CSAIL MIT-LCS-TR-921, (2003),
4. Baresi, L. and Spoletini, P.: On the Use of Alloy to Analyze Graph Transformation Systems. In, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, (eds.) Graph Transformations. Lecture Notes in Computer Science, vol. 4178, pp. 306-320. Springer Berlin / Heidelberg (2006)
5. Ehrig, K., Küster, J., and Taentzer, G., "Generating Instance Models from Meta Models", in Software and Systems Modeling, vol. 8: Springer Berlin / Heidelberg, (2009), pp. 479-500
6. EMF, "Eclipse Modeling Framework (Emf)". http://www.eclipse.org/modeling/emf/
7. Endresen, J., Carlson, E., Moen, T., Alme, K.-J., Haugen, Ø., Olsen, G.K., and Svendsen, A., "Train Control Language - Teaching Computers Interlocking", Computers in Railways XI (COMPRAIL 2008), Toledo, Spain, (2008)
8. Georg, G., Bieman, J., and France, R.B., "Using Alloy and Uml/Ocl to Specify Run-Time Configuration Management: A Case Study", in Workshop of the pUML-Group held together with the «UML»2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, A. Evans, R. B. France, A. M. D. Moreira, and B. Rumpe, Eds.: GI, (2001)
9. Gheyi, R., Massoni, T., and Borba, P., "A Theory for Feature Models in Alloy", in First Alloy Workshop. Portland, United States, (2006), pp. 71–80
10. Gogolla, M., Bohling, J., and Richters, M.: Validating Uml and Ocl Models in Use by Automatic Snapshot Generation. Software and Systems Modeling. 4, 386-398 (2005)
11. Herrmannsdoerfer, M., Ratiu, D., and Koegel, M., "Metamodel Usage Analysis for Identifying Metamodel Improvements", in SLE '10: 3rd International Conference on Software Language Engineering. Eindhoven, Netherlands: Springer, (2010)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, (2006)
13. Jacky, J.: The Way of Z: Practical Programming with Formal Methods. Cambridge University Press, (1997)
14. Kelly, S. and Pohjonen, R., "Worst Practices for Domain-Specific Modeling", in Software, IEEE, vol. 26, (2009), pp. 22-29
15. Kelly, S. and Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Inc., (2008)
16. Kelsen, P. and Ma, Q., "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. Toulouse, France: Springer-Verlag, (2008)
17. MOF, "The Metaobject Facility Specification". http://www.omg.org/mof/

18. Mougenot, A., Darrasse, A., Blanc, X., and Soria, M.: Uniform Random Generation of Huge Metamodel Instances. In, R. Paige, A. Hartman, and A. Rensink, (eds.). Lecture Notes in Computer Science, vol. 5562, pp. 130-145. Springer Berlin / Heidelberg (2009)
19. OCL, "Object Constraint Language". http://www.omg.org/spec/OCL/
20. OCL, "Ocl for Ecore". http://www.eclipse.org/modeling/mdt/?project=ocl
21. Sen, S., Baudry, B., and Mottu, J.-M.: Automatic Model Generation Strategies for Model Transformation Testing. In, R. Paige, (ed.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, vol. 5563, pp. 148-164. Springer Berlin / Heidelberg (2009)
22. Sen, S., Moha, N., Baudry, B., and Jézéquel, J.-M.: Meta-Model Pruning. In, A. Schürr and B. Selic, (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 5795, pp. 32-46. Springer Berlin / Heidelberg (2009)
23. Shah, S.M.A., Anastasakis, K., and Bordbar, B., "From Uml to Alloy and Back Again", in 6th International Workshop on Model-Driven Engineering, Verification and Validation, L. Lúcio and S. Weißleder, Eds. Denver, Colorado, USA: ACM, (2009)
24. Singh, N. and Babbar, R., "Build Metamodels with Dynamic Emf". http://www.ibm.com/developerworks/library/os-eclipse-dynamicemf/, (2007)
25. Straeten, R.V.D., "Alloy for Inconsistency Resolution in Mde", in The 8 th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2009), (2009)
26. Svendsen, A., Møller-Pedersen, B., Haugen, Ø., Endresen, J., and Carlson, E., "Formalizing Train Control Language: Automating Analysis of Train Stations", Comprail 2010, Beijing, China, (2010)
27. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling", Model Driven Engineering Languages and Systems (MoDELS 2008), Tolouse, France, (2008)
28. Wegmann, A., Lê, L.-s., Hussami, L., and Beyer, D., "A Tool for Verified Design Using Alloy for Specification and Crocopat for Verification", in First Alloy Workshop, D. Jackson and P. Zave, Eds. Portland, OR, USA, (2006)
29. Winkelmann, J., Taentzer, G., Ehrig, K., and Küster, J.M., "Translation of Restricted Ocl Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars", in Electronic Notes in Theoretical Computer Science (ENTCS), vol. 211. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., (2008)