OnTrack: An Open Tooling Environment for Railway Verification

Phillip James[1], Matthew Trumble[2,⋆],
Helen Treharne[2], Markus Roggenbach[1], and Steve Schneider[2]

[1] Swansea University, UK
[2] University of Surrey, UK

Abstract. OnTrack automates workflows for railway verification, starting with graphical scheme plans and finishing with automatically generated formal models set up for verification. OnTrack is grounded on an established domain specification language (DSL) and is generic in the formal specification language used. Using a DSL allows the formulation of abstractions that work for verification in several formal specification languages. Here, we demonstrate the workflow using CSP||B and suggest how to extend the tool with further formal specification languages.

1 Introduction

It is becoming common industrial practice to utilize Domain Specific Languages (DSLs) for designing systems [10]. Such DSLs offer constructs native to the specific application area. Formal methods often fail to be easily accessible for engineers, but designs formulated in DSLs are open for systematic and, possibly, automated translation into formal models for verification. DSLs also allow abstractions to be formulated at the domain level.

Considering the railway industry, defining graphical descriptions is the de facto method of designing railway networks. This enables an engineer to visually represent the tracks and signals etc., within a railway network. This paper describes OnTrack[1], an open tool environment allowing graphical descriptions to be captured and supported by formal verification. Our work is inspired by the SafeCap toolset [5] which is a graphical editor tailored towards Event-B analysis. In OnTrack, we emphasise the use of a DSL and decoupling this DSL from the verification method. The novelty of this is that we define abstractions on the DSL in order to yield an optimised description prior to formal analysis. Importantly, these abstractions allow benefits for verification in different formal languages. Our graphical editor can be used as a basis for generating different formal specifications in different languages. Such automated generation eliminates errors introduced when hand-coding formal specifications, improving for instance, the hand-coded specifications in [6,8,9]. Finally, OnTrack is designed for the railway domain, but the clear separation of an editor with support for abstractions from the chosen formal language is a principle more widely applicable.

⋆ The author was funded by an EPSRC vacation bursary, Summer 2012.

[1] OnTrack available for download from `http://www.csp-b.org`

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 435–440, 2013.

2 Workflow

Figure 1 shows the workflow that we employ in OnTrack. Initially, a user draws a *Track Plan* using the graphical front end. Then the first transformation, *Generate Tables* leads to a *Scheme Plan*, which is a track plan and its associated control tables. Control tables contain information about when routes can be granted, see [9] for details. Track plans and scheme plans are models formulated relative to our railway DSL meta-model, see Section 3. A scheme plan is the basis for subsequent workflows that support its verification. Scheme plans can be captured as formal specifications. This is achieved following two transformations: (1) a *Represent* transformation translates a *Scheme Plan* into an equivalent *Formal Scheme Plan* over the meta-model of the formal specification language (FSL) - this is the core transformation within the toolset; (2) various *Generate for Verification* transformations turn a *Formal Scheme Plan* into a *Formal Specification Text* ready for verification using external tools. These *Generate for Verification* transformations can enrich the models appropriately for verification. These transformations are validated via manual review.

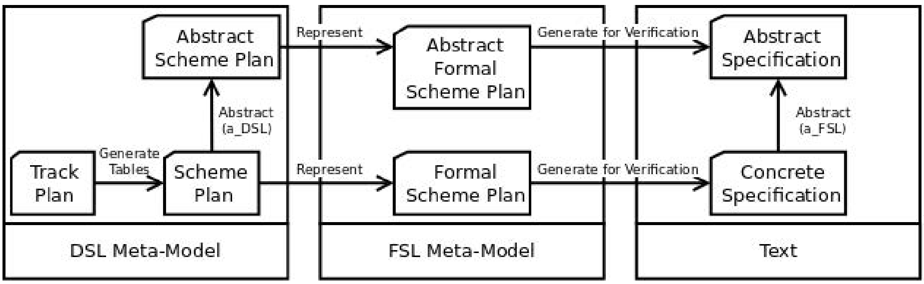

Fig. 1. OnTrack workflow

The horizontal workflow, described above, provides a validated transformation that yields a formal specification text that faithfully represents a scheme plan. In addition to this workflow, we are interested in abstractions to ease verification. Moller et al. [8] identify two abstractions: representing topological insights from the domain and reduction theorems over the language semantics. In OnTrack we define the topological abstractions with respect to the DSL, thus they are decoupled from the FSL. As any abstraction a_{DSL} w.r.t the DSL induces a corresponding abstraction a_{FSL} over specifications, it is possible to share them between different formal methods.

3 The OnTrack Editor

OnTrack implements the workflow from Section 2 in a typical EMF/GMF/Epsilon architecture [3,7]: a graphical editor realised in GMF is the front end for the user. As a basis for our tool, we have defined a modified version of the DSL developed by Bjørner [1]. The concepts of such a DSL can be easily captured within an ECORE meta-model which underlies our toolset. A small excerpt of topological concepts within our meta-model is given in Figure 2.

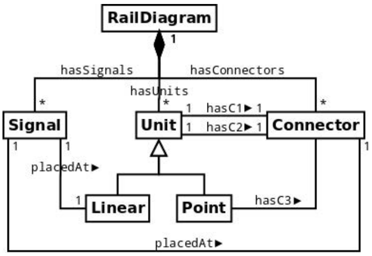

Fig. 2. Static concepts from Bjørner's DSL

A *Railway Diagram* is built from *Units*, *Connectors* and *Signals*. *Units* come in two forms: *Linear* representing straight tracks, or *Point* representing a splitting track. All *Unit(s)* are attached together via *Connector(s)*. Finally, *Signals* can be placed on *Linear* units and at *Connectors*.

Implementing a GMF front-end for this meta-model involves selecting the concepts of the meta-model that should become graphical constructs within the editor and assigning graphical images to them. Figure 3 shows the OnTrack editor that consists of a drawing canvas and a palette. Graphical elements from the palette can be positioned onto the drawing canvas. Within the editor, the Epsilon Wizard Language (EWL) for model transformations has been used to implement calls to the various scripts realizing different transformations. The first EWL wizard, *Generate Tables*, automatically computes a control table for a track plan. We omit details of this transformation and focus instead on the *Abstraction*, *Represent* and *Generate for Verification* transformations.

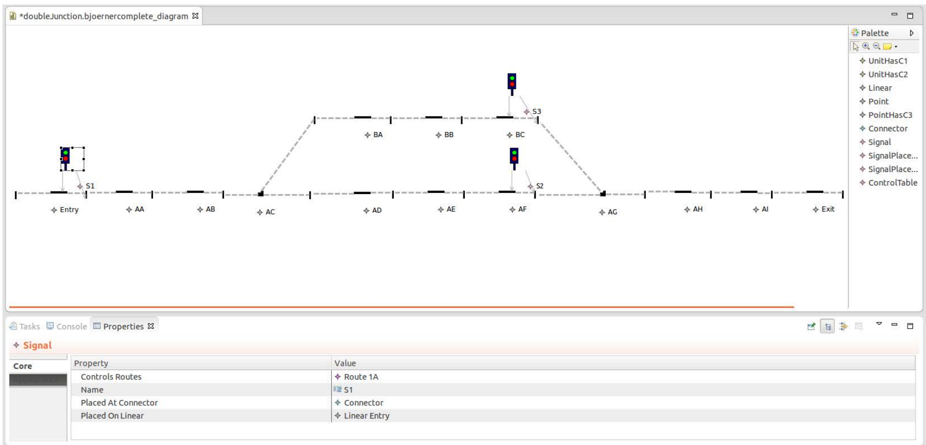

Fig. 3. A screenshot of "OnTrack" modelling a station

.

Listing 1.1. ETL rule for abstract model transformation

```
1  rule abs transform rd: Input!RailDiagram to rd2 : Target!RailDiagram {
2          rd.computeAbstractions();
3          for(ut:Unit in rd.hasUnits){
4              if(not (toDelete.contains(ut))){
5                  if(consToBeMapped.contains(ut.hasC1)) {
6                      ut.hasC1 = ut.hasC1.getMapping(); }
7                  if(consToBeMapped.contains(ut.hasC2)) {
8                      ut.hasC2 = ut.hasC2.getMapping(); }
9                  rd2.hasUnits.add(ut); } }
10         //Omitted code: similar computation with connectors and signals//
11         rd2.computeTables(); }
```

4 Automatic DSL Abstractions

We have implemented a particular a_{DSL} abstraction based on the *simplifying scheme plan* abstraction by Moller et al. [8]. Various sequences of units are "collapsed" into single units. This abstraction has been shown correct, and to improve the feasibility of verification [8]. The abstraction is implemented using the Epsilon Transformation Language (ETL) [7] that is designed for model transformations. Listing 1.1 gives an excerpt of our transformation. The algorithm uses the following list structures: `toDelete`: storing units to be removed and `consToBeMapped`: storing which connectors require renaming.

The `abs` rule performs as follows: line 1 states that the rule translates the given rail diagram `rd` to another `rd2`. The second line simply calls an operation `computeAbstraction()` on `rd` to compute which units can be collapsed and to populate the lists with appropriate values. For example, considering Figure 3, `toDelete` $= [AA, AB, BA, BB, AD, AE, AH]$. Next, the algorithm will consider every unit `ut` within `rd` (line 3). If `ut` is not in the list `toDelete` (line 4), then the algorithm will perform analysis on the connectors of `ut`. If connector one of `ut` is within the set of connectors requiring renaming (line 5), then the first connector of `ut` is renamed using a call to the operation `getMapping()` (line 6). Lines 7 to 8 of the algorithm perform these steps for connector two of `ut`. After this computation, the modified unit `ut` is added as an element to `rd2` (line 9). The algorithm continues in a similar manner, computing which connectors and signals should be added to `rd2`. Finally, an operation `computeTables` is called to compute a new control table for `rd2`. The result of this translation is that units `AA`, `AB`, `BA`, `BB`, `AD`, `AE` and `AH` are removed from the track plan in Figure 3.

5 Automatic Generation of CSP||B Models

Here we describe the implementation of the *Represent* and *Generate for Verification* transformations for CSP||B formal specifications. The use of CSP||B specifications for railway modelling is presented in [8,9].

Listing 1.2. One of the ETL rules for unit to CSP datatype transformation

```
rule processUnits transform u : Bjoerner!Unit to d : CSP!DataTypeItem {
    d.name = u.name;
    d.type = pos;
    if (pos_list.firstItem.isDefined()) {
        d.preceeds = pos_list.firstItem; }
    pos_list.size = pos_list.size + 1;
    pos_list.firstItem = d; }
```

The goal of the *Represent* transformation is to iterate through a scheme plan, which is an instance of our DSL meta-model, in order to produce instances of the CSP||B meta-models. It is implemented using ETL. CSP||B meta-model instances contain collections of objects required to produce the final specification text. They do not include information on the structure of statements for the final formal specification. The Epsilon Validation Language (EVL) [7] can be used to validate all required objects are defined as expected. We achieve traceability between the meta-models by defining a structured ETL transformation, i.e., providing separate ETL scripts that reflect the final specification text architecture. Overall, our CSP||B model consists of six specifications [9], each generated by a separate ETL script. These scripts consist of 16 rules, 1 local operation and a 17 shared operations. Listing 1.2 gives an example rule that transforms units of a scheme plan to a CSP data type. For each unit, the `processUnits` rule constructs a corresponding `DataTypeItem` which is then added to the datatype (`pos_list`). For example, for Figure 3 `pos_list` = [AA,AB,AC,. . .].

The *Generate for Verification* transformation translates CSP||B meta-model instances into formal specification text. Interestingly, in CSP||B the formal specification text differs depending on the property to be proven, see [8] for details. Therefore, the *Generate for Verification* transformation produces a number of different specification texts. These transformations are implemented using the Epsilon Generation Language (EGL) [7] for generating text. For example, the `pos_list` datatype instance becomes the following fragment of CSP: `datatype ALLTRACK = AA | AB | AC |` . . .. The transformations are novel as they apply pre-processing using Apache Velocity Java templates to avoid code repetition. These together with the EGL are used to generate models. Note that the CSP||B instance models produced from the *Represent* transformation only contain the information of a scheme plan. They do not include a model of the interlocking algorithm. This algorithm remains constant for all scheme plans and is therefore defined in a template file which is used in the *Generate for Verification* transformation to enrich the CSP||B specifications. Similarly, the behavioural description of a train remains constant and is again defined in a separate template file. Overall, we define six templates which reflect the final CSP||B architecture (1 CSP script and 5 B machines, see [9] for details). This gives a clear correspondence between the templates and formal specification structure.

6 Lessons Learnt and Discussion

The OnTrack toolset achieves the aim of automating the tedious production of formal specifications. The toolset allows for abstractions to be defined over the DSL in order to produce optimised railway models, from which transformations to formal specifications can be defined. Importantly, these abstractions are decoupled from the formal specifications. In building the tool, the encoding of the DSL into a meta-model is straightforward, however there needs to be a close relationship between the graphical artifacts and the meta-model. The benefit of using the toolset is that we can focus our efforts on understanding the impact of the verification results on the safety of a scheme plan.

Current work includes the development of a *G*enerate for Verification transformation to the algebraic specification language CASL. Manual encoding has shown the abstractions over the DSL also aid verification for a CASL based railway modelling approach [6].

In order to extend the tool to produce formal specifications in languages other than CSP||B, e.g., for railway verification based on NuSMV by [4], the following would be required: define a meta-model for the chosen formal language and then define the *Represent* and *Generate for Verification* transformations for that language.

Future improvements that would aid our understanding of the results is to visualise feedback of any counterexamples, produced during verification on the scheme plan itself. Similar visualisations have already been achieved in [2].

References

1. Bjørner, D.: Formal Software Techniques for Railway Systems. In: CTS 2000 (2000)
2. dos Santos, O.M., Woodcock, J., Paige, R.F.: Using model transformation to generate graphical counter-examples for the formal analysis of xUML models. In: ICECCS, pp. 117–126. IEEE Computer Society (2011)
3. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional (2009)
4. Haxthausen, A.E.: Automated generation of safety requirements from railway interlocking tables. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part II. LNCS, vol. 7610, pp. 261–275. Springer, Heidelberg (2012)
5. Iliasov, A., Romanovsky, A.: SafeCap domain language for reasoning about safety and capacity. In: Workshop on Dependable Transportation Systems. IEEE CS (2012)
6. James, P., Roggenbach, M.: Designing domain specific languages for verification: First steps. In: ATE 2011. CEUR (2011)
7. Kolovos, D., Rose, L., Paige, R., García-Domínguez, A.: The Epsilon Book. The Eclipse Foundation (2012)
8. Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Defining and Model Checking Abstractions of Complex Railway Models using CSP||B. In: HVC 2012. LNCS (to be published)
9. Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Railway modelling in CSP||B: the double junction case study. In: AVOCS 2012. EASST (2012)
10. Invensys Rail: Invensys Rail Data Model – Version 1A (2010)