Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Modellierung und Verifikation von Eisenbahnsystemen: Übersetzung von RailML in die B-Methode

Modeling and Verification of Railway Systems:
Translation of RailML Into the B-Method

Masterarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Jan Gruteser

Beginn der Arbeit:	02. Mai 2023
Abgabe der Arbeit:	31. Oktober 2023
Erstgutachter:	Prof. Dr. Michael Leuschel
Zweitgutachter:	Dr. Jens Bendisposto

Abstract

The aim of this work is to translate files written in railML to the formal B-method to enable formal verification and validation of the specifications. RailML is an XML-based format designed to facilitate the exchange of information about railway systems between railway applications. Since undetected small errors in the modelling can lead to serious errors in the real system, especially with safety-critical properties such as train protection, the files should always be subjected to an automatic validation.

The approach developed in this thesis allows automatic syntactic and semantic validation against predefined rules. For this purpose, ProB and its integrated B-Rules DSL are used to validate these rules. In addition to the validation, a B-model is presented to animate the behaviour of the specification, which can also be used for simulations and statistical tests using SimB. By using generated machines, the user can also define custom rules for validation using the mathematical expressiveness of the B language. A strategy for creating visualisations of the topology using Graphviz is presented, which can also be used with VisB to visualise the current state. The entire process is made available in a user-friendly way through integration into the ProB2-UI.

Finally, case studies are investigated to show that the implemented validation process can be efficiently applied to complex models and that errors can be successfully detected in some of the models.

Acknowledgements

I would like to thank several railML partners for their support. Firstly, railML.org, who helped me a lot during the development by providing the Advanced Example. Then the Norwegian Jernbanedirektoratet together with trafIT solutions, for the inspiring exchange, but also for providing access to further example files in railOscope in addition to those of the Norwegian network. And also NEAT, who gave me the opportunity to create my own railML examples with their D4R Track Planner. Without their help, the realisation of this thesis would certainly have been more difficult.

I am also grateful to my supervisor Michael Leuschel for his good support, helpful discussions and the addition of features to ProB that have been very useful for my work. I would also like to thank Fabian Vu for his support with SimB.

Contents

1 Introduction

Undetected small errors in safety-critical systems may cause serious accidents. Therefore, when developing such systems, it is essential to prevent these errors from the beginning and build correct software by construction. Formal methods can effectively help to achieve this goal. They enable a mathematical description of the system's properties and requirements, and their automatic (and manual) verification. This is often done using so-called state-based methods, where the current state of the system is considered and, based on this, various events can occur that lead to further states. Using model checking, it is then possible to traverse through the entire state space, check the properties in each state, and determine whether a safety-critical state exists. A prominent example for a state-based method is the B-method developed by Abrial [Abr96], which is now widely used in both research and industry. Particularly in the railway sector, the B-method has become established for verification of railway (interlocking) systems, but in recent decades also for validation of data from external sources, such as XML files. The B-method and some of its applications in railways are described in Section 1.1.

The aim of this work is to convert railway system specifications described in the Railway Markup Language (railML®)[CNC17; Nas+04; Rai23a] to a representation in the B-method to allow formal verification and validation of the specified system. This is done using ProB [LB03], an animator, constraint solver, and model checker for the B-method. First, the data is read from the railML file using an external function and then imported using ProB's rule validation language [HSL16]. Syntactic and semantic validation is performed on the imported data and new data structures are derived. Based on this, the model can be animated and simulated with SimB [VLM21] using a formal B model developed for this purpose in classical B. The imported data can also be stored and animated in separately generated machines. These steps can be carried out in a user-friendly way by a plugin currently being developed for the ProB2-UI [Ben+21]. It also enables the creation of configurable visualisations of the topology compatible with VisB [WL20].

The structure of this thesis is based on the sequence of the overall process. After a short introduction to the B-method and railML 3, Section 2 introduces the language for rule validation in ProB. Section 3 deals with the import process of railML data into ProB, including syntactic and semantic validation. Section 4 then goes into the representation of the data in B and the following Section 5 explains how the animation based on this is implemented in B. After this, Section 6 elaborates on the advantages on generation of standalone B machines out of the imported data and describes the implementation. Simulation using SimB (Section 7) and visualisation using VisB (Section 8) complete the investigation of the animation model. The following Section 9 presents the current approach for integrating the developed process into a plugin for the ProB2-UI. Finally, in Section 10, results for exemplary case studies are discussed and then performance evaluations for different models are carried out (Section 11).

1.1 The B-Method in Railway Context

In general, formal methods are of great interest in the railway sector and are recommended by the EN50128 standard for SIL 3 and SIL 4 systems. Ferrari and Beek [FB22] conducted research on formal techniques used in the railway industry and found that B is one of the dominant languages for formal modelling. The most common techniques identified were model checking at 47% and simulation at 27%, both of which are applied in this work. This is done using ProB, which is a commonly used tool in this context, appearing in 9% of the studies examined.

The B-method has already been successfully applied to industrial railway projects in particular, a famous example being the driverless Paris metro line 14, for which all safety-critical components of the system were formally developed using B and which has now been operating safely since 1998 [But+20]. There are many other projects using B and its successor Event-B, including, for example, a real-time demonstration of ETCS Hybrid Level 3 using ProB [Leu23].

Originally the classical B-method [Abr96] has been developed for creation of safety-critical software by formal specifications based on set theory and first-order logic. Abstract machines are refined by adding more details in each refinement step until a concrete specification is reached which can then be used for code generation. A machine in classical B can basically have the following components: `CONSTANTS` that are described by `PROPERTIES`; `SETS` for the definition of new types; `VARIABLES` that must always fulfil certain `INVARIANT`s and whose initial values are set in the `INITIALISATION`; `OPERATIONS` that change the state, i.e. the variable assignment, and can only be executed if their preconditions (guards) are fulfilled. In ProB, `DEFINITIONS` can also be used, which, among other things, allow access to external functions (e.g. for loading XML files). These formal B models can then be animated, model checked, visualised, and also simulated to verify that the specification fulfils the (safety) requirements and allow domain experts to investigate whether the specified model behaves as expected. To enable this for the models specified in railML, a B model has been developed based on the simple interlocking system model by Abrial [Abr10].

Besides formal modelling of systems, formal data validation has emerged as another useful application area of the B-method in recent decades, also with several successful projects [But+20; PK21]. The aim is to validate static input parameters (constants) of a specification against previously defined requirements and assumptions. Ordinary B machines, but also domain-specific languages, can be used to formulate the rules that formally capture these assumptions [HSL16; LM16; Lec+17]. Here, this approach is used to formally validate the railML data against syntactic and semantic rules formulated using ProB's built-in rule validation language before animating the system, which is documented in detail in Section 2. Since this translates the rules into an internal representation using classical B (cf. Appendix E), all machines are written in the classical B notation.

1.2 RailML 3

The purpose of railML is to enable a standardised data exchange between different railway applications. Based on XML, it has four main sub-schemas: Infrastructure (`IS`), Rollingstock (`RS`), Timetable (`TT`), and as of railML 3 also Interlocking (`IL`). RailML 3 uses a completely new approach based on the RailTopoModel (RTM) [Rai23c], which is not backward compatible with railML 2. Since interlocking data is of great interest for verification of safety properties with the formal B-method, this work focuses on the current version of railML 3, namely railML 3.2. The following section provides a brief introduction to the parts of railML 3 that are later used in the B model. How these elements are modelled in B is described in Section 4. Due to the high level of detail of the entire railML schema, only a subset of all properties is implemented in this work. The complete documentation of the railML 3 schema can be obtained from the railML 3 wiki [Rai23b]. Some general concepts of railML 3 and the integrated RTM are addressed by Kolmorgen et al. [Kol+23].

In some places there are references to the *Simple Example*, the *Advanced Example*, or *Norwegian examples*. These refer to railML sample files that were used as case studies to test the implementation and are introduced in more detail in Section 10.

Topology The entire model of railML 3 is based on the RTM, which is a multi-level approach for definition of topologies. It uses a basic node-edge-model for description of connected elements on a certain level of the topology. In railML 3, the nodes are so called `netElements` which are connected by edges defined in `netRelations`. In a further section `networks` all topology elements can be assigned to different networks and within them to one of the three level types (`micro`, `meso`, and `macro`) that are given by the RTM. Elements at micro level are closely related to the infrastructure, such as net elements that form a track or contain signal locations. Meso and macro level elements describe a coarser view of the topology, such as connections between operational points or stations. The B model in its current form aims to use only elements that are located at micro levels, as the model is intended to investigate the infrastructure and interlocking data, which mostly happens at micro level. Also, the grouping of net elements is currently not considered by the formal model, as these properties are only relevant for grouping lower level elements to connect them to a higher level element. Further details on the linking of net elements in the RTM and differences between the levels are elaborated by Bollig [Bol20], Hlubuček [Hlu17], and Wunsch and Jaekel [WJ17] (in German).

In railML 3, the topology data is specified in the `topology` section of the `infrastructure` schema[1]. An example of a simple definition is given in Listing 1, which encodes the topology shown in Figure 1. Each of the specified `netElements` induces a corresponding system of intrinsic coordinates ranging from 0.0 to 1.0, where a coordinate indicates the position relative to the endpoints of the `netElement`. A `netRelation` connects the endpoints of

[1]`https://wiki3.railml.org/wiki/IS:topology` (accessed on 22/10/2023)

two different `netElements` as given by the attributes `positionOnA/B`. They can take the two values 0 and 1 to specify the intrinsic end coordinate to which the relation should be connected. For example, the two coordinates (`ne02`, 1.0) and (`ne04`, 0.0) are connected by a relation. For each relation, it is also possible to provide information about the usable direction of a relation by setting `navigability`. In the example of Listing 1 there is a triangular relationship between `ne02`, `ne03` and `ne04`. As the relation between `ne03` and `ne04` is non-navigable (line 23), there must be a switch between `ne02`, `ne03`, and `ne04`. Finally, all elements are declared to be a `networkResource` of the microscopic level `lv01`.

Listing 1: Simple Topology Encoded in RailML 3

```
 1: <railML version="3.2">
 2:     <infrastructure id="is01">
 3:         <topology>
 4:             <netElements>
 5:                 <netElement id="ne01"/>
 6:                 <netElement id="ne02"/>
 7:                 <netElement id="ne03"/>
 8:                 <netElement id="ne04"/>
 9:             </netElements>
10:             <netRelations>
11:                 <netRelation id="nr0102" positionOnA="1" positionOnB="0"
                        navigability="Both">
12:                     <elementA ref="ne01"/>
13:                     <elementB ref="ne02"/>
14:                 </netRelation>
15:                 <netRelation id="nr0203" positionOnA="1" positionOnB="1"
                        navigability="Both">
16:                     <elementA ref="ne02"/>
17:                     <elementB ref="ne03"/>
18:                 </netRelation>
19:                 <netRelation id="nr0204" positionOnA="1" positionOnB="0"
                        navigability="Both">
20:                     <elementA ref="ne02"/>
21:                     <elementB ref="ne04"/>
22:                 </netRelation>
23:                 <netRelation id="nr0304" positionOnA="1" positionOnB="0"
                        navigability="None">
24:                     <elementA ref="ne03"/>
25:                     <elementB ref="ne04"/>
26:                 </netRelation>
27:             </netRelations>
28:             <networks>
29:                 <network id="nw01">
30:                     <level id="lv01" descriptionLevel="Micro">
31:                         <networkResource ref="ne01"/>
32:                         <networkResource ref="ne02"/>
33:                         <networkResource ref="ne03"/>
34:                         <networkResource ref="ne04"/>
35:                         <networkResource ref="nr0102"/>
36:                         <networkResource ref="nr0203"/>
37:                         <networkResource ref="nr0204"/>
38:                         <networkResource ref="nr0304"/>
39:                     </level>
40:                 </network>
41:             </networks>
42:         </topology>
43:     </infrastructure>
44: </railML>
```

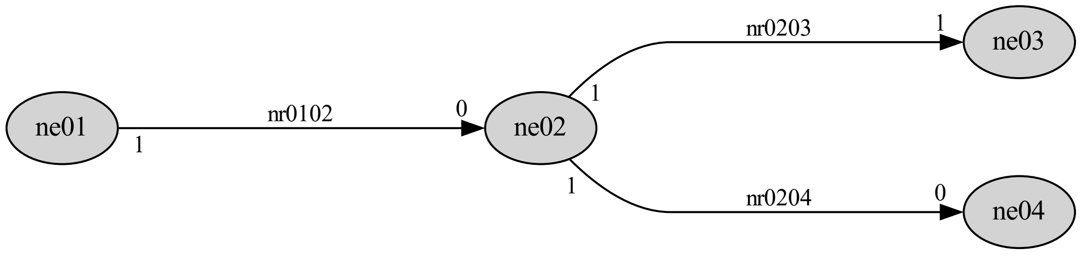


Figure 1: Graph Visualisation of the Example Topology

Infrastructure Elements After specifying a topology, infrastructure elements can be defined that are located in the topology. This is done in the section `functionalInfrastructure`, where all infrastructure elements can be specified grouped by type. Basically, an ID and a position must be provided for each element. Further properties can be provided through attributes or child elements, which may also contain additional details. The most important elements for use in the B model are all movable elements (i.e. derailers, crossings and switches), tracks, train detection elements and signals. As an example, Listing 2 defines a switchable train movement signal with the name "`69W04Y`" located at the net element `"ne_b05"` (taken from the "Simple Example" by railML.org).

Listing 2: Signal Encoded in RailML 3 on Infrastructure Side

```
<functionalInfrastructure>
    <signalsIS>
        <signalIS id="sig08" isSwitchable="true">
            <name name="69W04Y" language="en"/>
            <spotLocation id="sig08_sloc01" netElementRef="ne_b05"
                applicationDirection="normal" pos="100.0"/>
            <isTrainMovementSignal/>
        </signalIS> [...]
    </signalsIS> [...]
</functionalInfrastructure>
```

The RTM allows to specify the location of infrastructure elements pointwise, linearly or as an area. For specifying location data, three corresponding types can be used in railML 3, namely `areaLocation`, `linearLocation`, and `spotLocation`. The `areaLocation`-type is currently not yet used in the B model since areas occur very rarely when describing locations on a microscopic level. The type `linearLocation` can be used to indicate the location of an element that extends linearly over several net elements like a track or platform. It is also possible to use several parallel linear sections, for example for tunnels. To specify a single location, as for the signal in Listing 2, the type `spotLocation` is used. Each location consists of a net element in conjunction with an intrinsic coordinate on that element.

The infrastructure elements are defined independently of each other, which means they usually do not refer to each other and mainly describe the location, the physical characteristics and capabilities of the element. Behaviour, dependencies and references are introduced in the interlocking logic, which is why there is often one type for each of the schemas.

Interlocking The interlocking schema defines the interlocking logic for assets in its `assetsForInterlockings` section. Important concepts are track vacancy detection sections (`tvdSections`) for information about the occupation of track sections between certain `trainDetectionElements` and `routes` that can be reserved by trains. A `route` specification looks like the example in Listing 3. A `routeEntry` and a `routeExit`, usually a reference to an interlocking signal, must be specified and required switch positions for the uniqueness of the connected path can be specified. Similarly, TVD sections can be linked to the route and overlaps to the route exit. An `overlap` secures an additional section after the route exit. Further safety requirements such as flank protection for a route can be specified by `routeRelations`.

Listing 3: Route Encoded in RailML 3

```
<route id="rt1">
    <facingSwitchInPosition inPosition="right">
        <refersToSwitch ref="il_swi1"/>
    </facingSwitchInPosition>
    <routeEntry id="ren_rt1">
        <refersTo ref="il_sig1"/>
    </routeEntry>
    <hasTvdSection ref="tvd1"/>
    <routeExit id="rex_rt1">
        <refersTo ref="il_sig2"/>
        <hasOverlap ref="ovl1" />
    </routeExit>
</route>
```

The assets for interlocking include the already mentioned twin elements on interlocking side for the infrastructure elements, which is especially the case for the movable elements. For example, there is exactly one `switchIS` element and one `switchIL` element referencing the infrastructure element for each ordinary switch. On the interlocking side the branches from the track perspective, the throw times and any associated sections for controlling the track vacancy are specified, while on the infrastructure side mainly the position of the switch and its branches in the topology are described.

There are other subsections in the interlocking scheme. In the `signalBoxes` section, properties of the signal boxes themselves can be defined, for example the behaviour of the signals controlled by signal plans (`implementsSignalplan`) having different `aspectRelations`. The `specificInfrastructureManager` section contains further infrastructure manager related properties such as the possible signal aspects or reset strategies of TVD sections. For signals, for instance, the individual aspect is mapped by `hasAspect` to a generic aspect, which can then be used in the formal modelling.

Some further details, including the other railML schemas (e.g. for visualisation), will be introduced as their use in the B-model is explained.

2 The Rule Validation Language in ProB

In addition to modelling the behaviour of system specifications, formal methods have also evolved towards data validation. The objective is to validate static properties, which may originate from external sources such as XML or CSV files, against previously developed rules. According to Lecomte et al. [Lec+17], this type of verification was still done manually around the year 2000, which is a very error-prone and time-consuming process. The use of formal methods, including the B-method, considerably simplifies this process, as the rules can be formally coded and usually be verified automatically within a much shorter time. In order to enable easy access to the capabilities of the B language for formulation of such rules by domain experts, domain-specific languages (DSL) have been developed. Lecomte et al. [Lec+17] also provide an example of a "verification rule" (Fig. 4).

ProB has its own rule-based language for convenient data validation based on classical B [HSL16; Hei18], which has similarities to the example mentioned above. By formulating so-called rules and computations, it is possible to check the rules and compute variables depending on success or failure of rules. The first section of this chapter begins with a detailed introduction to the rule validation language implemented in ProB, which has been similarly added to the ProB documentation[2] as part of this project. This is followed by a description of how the mechanisms for loading and validating XML files can be applied.

2.1 General Usage

The rule validation language, hereafter referred to as B-Rules DSL, mainly provides operations for data validation. *Rules* allow checking for expected properties, while *computations* can be used to define and compute variables based on the successful execution of certain rules. Furthermore, *functions* allow values to be computed multiple times depending on different inputs. Rules machines are stored in `.rmch`-files. The general setup for the machine header is:

```
RULES_MACHINE machine_name REFERENCES list of rules machines.
```

The latter allows the inclusion of other rules machines and ordinary B machines that contain only constants, but not yet any other B machines. Below, `SETS`, `DEFINITIONS`, `PROPERTIES` or `CONSTANTS` can be used as in a normal B machine. Note that `VARIABLES` are not allowed as they are set by rule based computations.

Rules Rules can be defined in the `OPERATIONS`-section of a rules machine. Depending on whether the expectations are met, a rule returns `SUCCESS` or `FAIL`. If a rule fails, additionally provided string messages are returned as counterexamples. The general structure of a rule in the B-Rules DSL is presented in Listing 4.

[2] `https://prob.hhu.de/w/index.php?title=Rules-DSL` (accessed on 25/10/2023)

Listing 4: General structure of a RULE in B-Rules DSL

```
RULE rule_name
DEPENDS_ON_RULE list of rules
DEPENDS_ON_COMPUTATION list of computations
ACTIVATION predicate
ERROR_TYPES positive number of error types
BODY
    arbitrarily many rule bodys (see below)
END
```

The specified `rule_name` will be the name of the operation and variable storing the result. If a rule depends on other rules, it can only be executed if the specified rules have been successfully checked, i.e. their corresponding variable `rule_name` has the value `SUCCESS`. In addition, rules can depend on computations. In this case, a rule is enabled when the specified computations have been executed. If a rule uses variables that are defined by computations, the corresponding computations are added implicitly as dependencies and do not have to be declared explicitly. Any other preconditions can be specified as an `ACTIVATION` predicate. An important note is that the activation predicate is evaluated statically at initialisation and disables the rule if the predicate is false. Activation predicates and dependencies can be omitted if they are not needed.

To use different error types (for example, if a rule has multiple bodies and it is necessary to distinguish between them), the number of error types has to be declared in the rule header. Error types are also optional.

The actual rule conditions are specified within the body of a rule, which contains the name and the preconditions. A rule succeeds if and only if all rule conditions in its body are satisfied. There are two constructs for rule bodies that can be used arbitrarily often in the body of a rule. The one shown in Listing 5 is formulated in a positive way, i.e. the execution of the rule leads to `SUCCESS` if the conditions in the `EXPECT`-part are fulfilled.

Listing 5: General structure of the RULE_FORALL body

```
RULE_FORALL
    list of identifiers
WHERE
    conditions on identifiers
EXPECT
    conditions that must be fulfilled for this rule
ERROR_TYPE
    number encoding error type, must be in range of error types
COUNTEREXAMPLE
    STRING_FORMAT("errorMessage ~w", identifier from list)
END
```

Alternatively, a negated rule can be formulated, which is shown in Listing 6. Here the execution of the rule results in `FAIL` if the conditions in the `WHEN` part are fulfilled.

Listing 6: General structure of the RULE_FAIL body

```
RULE_FAIL
    list of identifiers
WHEN
    conditions on identifiers for a failing rule
ERROR_TYPE
    number encoding error type, must be in range of error types
COUNTEREXAMPLE
    STRING_FORMAT("errorMessage ~w", identifier from list)
END
```

For both, the counterexamples are of the type `INTEGER <-> STRING`. The integer contains the error type, while the string contains the message of the counterexample.

Computations Computations can be used to define variables. As for rules, their activation can depend on further rules, computations or any other predicate specified as an activation condition. Again, the activation condition is evaluated at initialisation and sets the computation status variable to `COMPUTATION_DISABLED` if the predicate is false. Furthermore, a `DUMMY_VALUE` can be set, which initialises the variable with the specified value instead of the empty set before execution of the computation. This mechanism implies that each variable defined by a computation must be a set of type `POW(S)` for any type `S`. A computation can be replaced by a previously defined computation if it sets the same variable (of the same type) by using `REPLACES`. The general syntax for computations is shown in Listing 7.

Listing 7: General structure of a COMPUTATION in B-Rules DSL

```
COMPUTATION computation_name
DEPENDS_ON_RULE list of rules
DEPENDS_ON_COMPUTATION list of computations
ACTIVATION predicate
REPLACES identifier of exactly one computation
BODY
    DEFINE variable_name
        TYPE type of variable
        DUMMY_VALUE value of variable before execution
        VALUE value of variable after execution
    END
END
```

Activation predicates, dependencies, and also the dummy value can be omitted if they are not needed. After the execution of a computation, the value of the corresponding variable `computation_name` is changed from `NOT_EXECUTED` to `EXECUTED` and the variable `variable_name` has the value `VALUE`. For related computations, it may be useful to use multiple `DEFINE` blocks in one computation. Separated by "`;`", the body of a computation can contain any number of variable definitions.

Functions Functions formulated as in Listing 8 can be called from any rules machine that references the machine containing the function. Depending on input parameters that must satisfy specified preconditions, the function returns output value(s) that must satisfy optional postconditions. In the body, any B statement can be used to (sequentially) compute the output value.

Listing 8: General structure of a FUNCTION in B-Rules DSL

```
FUNCTION output <-- function_name(list of input parameters)
PRECONDITION predicate
POSTCONDITION predicate
BODY
    output := ...
END
```

Additional Syntax There are some useful predicates available in rules machines that can be used to check the success or failure of rules. It is also possible to check whether a certain error type was returned by a rule. These are:

- `SUCCEEDED_RULE(rule1)`: `TRUE`, if the check of rule1 succeeded
- `SUCCEEDED_RULE_ERROR_TYPE(rule1,1)`: `TRUE`, if the check of rule1 did not fail with error type 1
- `GET_RULE_COUNTEREXAMPLES(rule1)`: set of counterexamples of rule1
- `FAILED_RULE(rule1)`: `TRUE`, if the check of rule1 failed
- `FAILED_RULE_ERROR_TYPE(rule1,2)`: `TRUE`, if check of rule1 failed with error type 2
- `FAILED_RULE_ALL_ERROR_TYPES(rule1)`: `TRUE`, if the check of rule1 failed with all possible error types for rule1
- `NOT_CHECKED_RULE(rule1)`: `TRUE`, if rule1 has not yet been checked
- `DISABLED_RULE(rule1)`: `TRUE`, if rule1 is disabled (its preconditions are not fulfilled)

Another functionality of rules machines are `FOR`-loops. Listing 9 illustrates an example where the rule always fails and returns

`{1 |-> "example_rule_fail: 2", 1 |-> "example_rule_fail: 3"}`.

Listing 9: Example of a `FOR`-loop in B-Rules DSL

```
RULE example_rule
BODY
    FOR x,y IN {1 |-> TRUE, 2 |-> FALSE, 3 |-> FALSE} DO
        RULE_FAIL WHEN y = FALSE
        COUNTEREXAMPLE STRING_FORMAT("example_rule_fail: ~w", x)
    END
END
```

Internal Representation Each rules machine is internally translated to an ordinary B machine, which can be accessed as its internal representation. The translation of an example rule can be found in Appendix E. Another comparison of a rules machine and its internal machine is illustrated by Hansen et al. [HSL16].

Include Rules Machines into other Projects Currently, it is not possible to include rules machines directly into any other machines. Instead, the beforehand described internal representation of the hierarchical topmost rules machine must be saved as `.mch`-file. After changing the machine name accordingly, the rules can be included and used via this machine. This is discussed further in Section 3 on the use of rules machines in this project.

Plugin for ProB2-UI A plugin is available for the ProB2-UI from Heinzen [Hei18] for convenient access to the functionality of rules machines. It allows execution of rules and computations together with its dependencies, and lists the counterexamples in a separate tab. For use in this project, the code had to be slightly adapted to make the plugin compatible with the current version of ProB2-UI.

2.2 Import and Validate XML Files

The concept of rule validation can be used to validate data from an XML file and load the validated data into ProB by successive computations. A brief description of importing XML files in ProB is also given by St-Denis [St-23]. In general, XML data can be loaded in ProB by the external function `READ_XML`. It takes the relative path to the XML file to be imported as the first argument and the encoding of the file as the second argument. Valid encodings are listed in `LibraryXML.def`, whereby `"auto"` should be selected by default, which tries to determine the encoding based on the header of the XML file. `READ_XML` returns a sequence of XML elements of the type `XML_Element_Type` as shown in Listing 10.

Listing 10: Type of an XML Element loaded by `READ_XML`

```
XML_Element_Type ==
    struct(
        recId: NATURAL1,
        pId: NATURAL,
        element: STRING,
        attributes: STRING +-> STRING,
        meta: STRING +-> STRING )
```

Each element record has a unique identifier `recId`, which is also its position in the sequence, and stores the relationship to its parent in `pId`. Furthermore, the type of the element is stored as `STRING` in the field `element` and all existing attributes are available in `attributes` as a function from the attribute type to the value of the attribute. Finally, additional meta data such as line numbers is made accessible via the field `meta`.

Note that `attributes` by definition excludes multiple values for the same attribute types (which is also an important property to validate for an XML file), but also only includes existing attributes. Since optional attributes are common in XML schemas, this also raises the problem of how to deal with missing attributes that are not known when the XML file is parsed. For attributes with default values this can easily be avoided without changing the type by using the default value. To avoid this issue for variables without default value, the data can be encapsulated in a set. If the attribute is present, the set contains its value, otherwise the set is empty. Listing 11 illustrates the described technique with a basic example.

Listing 11: Example for Import and Validation of an XML File in B-Rules DSL

```
RULES_MACHINE XML_import
DEFINITIONS
    "LibraryXML.def"
CONSTANTS
    xml_data
PROPERTIES
    xml_data = READ_XML("xml_file.xml", "auto")
OPERATIONS
    COMPUTATION set_version
    BODY
        DEFINE version
        TYPE FIN(STRING)
        VALUE IF "version" : dom(xml_data(1)'attributes) THEN
           {xml_data(1)'attributes("version")} ELSE {} END
    END;
    RULE is_valid_version
    DEPENDS_ON_COMPUTATION set_version
    BODY
        RULE_FAIL v
        WHEN
            v : version & v /: {"3.1","3.2"}
        COUNTEREXAMPLE
            "xyz of version "^v^" is currently not supported"
        END
    END
END
```

An XML file is read and stored in the constant `xml_data`. A computation `set_version` is used to extract the version of the used XML schema, which is then checked by the rule `is_valid_version`.

The following chapter describes the approach of how the rules machines are used to import and validate railML data, and how the imported data can be accessed in B.

3 The Import Process

The techniques of the B-Rules DSL described in Section 2 are used to import and validate railML data in ProB. In this way, the entire conversion process can be handled on the B-side, enabling import using ProB without any additional dependencies. It also allows direct use of ProB's constraint-solving capabilities to compute logical relationships directly from the data. This was also the reason for the decision to use the B-Rules DSL, whose computations allow a sequential import of data using ProB and the B-method. The resulting machines can be used standalone by experienced users. Furthermore, a plugin for the ProB2-UI is currently under development for a more convenient and interactive use, which is described in detail in Section 9.

The import and validation process is based on several rules machines, where each machine is associated with one of the subschemas `Infrastructure`, `Interlocking`, `Visualization`, and `Common`. Each of them loads the railML data into B-records to ensure type checking of the attributes and then performs syntactic validation of the loaded data for the railML 3-schema. After syntactic validation, the data from the records is transferred into data structures that allow easier access to certain fields and also map relationships between different elements, such as the path of routes. Then, additional rules are executed to validate semantic constraints for both imported and derived data.

The used rules machines are linked by referencing and finally merged into one machine that simultaneously imports and validates the RailML data, as illustrated in Figure 2. In RailML 3, almost all data depends on the infrastructure data (`IS`), which is also reflected in the hierarchy of the rules machines. Also at the lowest level is the independent machine for the schema common (`CO`), which provides individual names of elements and some general properties such as speed profiles. Since interlocking (`IL`) and visualisation (`VIS`) data depend on infrastructure and common data, both machines reference the corresponding machines.

The following subsections describe the import process in detail and how the validated data is further processed for animation.

3.1 File Import

The starting point is the rules machine `RailML3_readFile.rmch`, where the raw import of XML data into B is done by the external function `READ_XML` as introduced in Section 2.2. Here, the read data is stored in a constant `data`, it is checked whether the XML file is of type `railML` at all and the railML version is extracted into a constant `version`. Currently supported versions are 3.1 and 3.2, any other version will be rejected by an associated rule. Both versions differ only in a few places that are relevant for the conversion to B. Storing the version allows the model to use case distinctions where appropriate later, so that the model supports both versions at the same time without the need for an adaption.

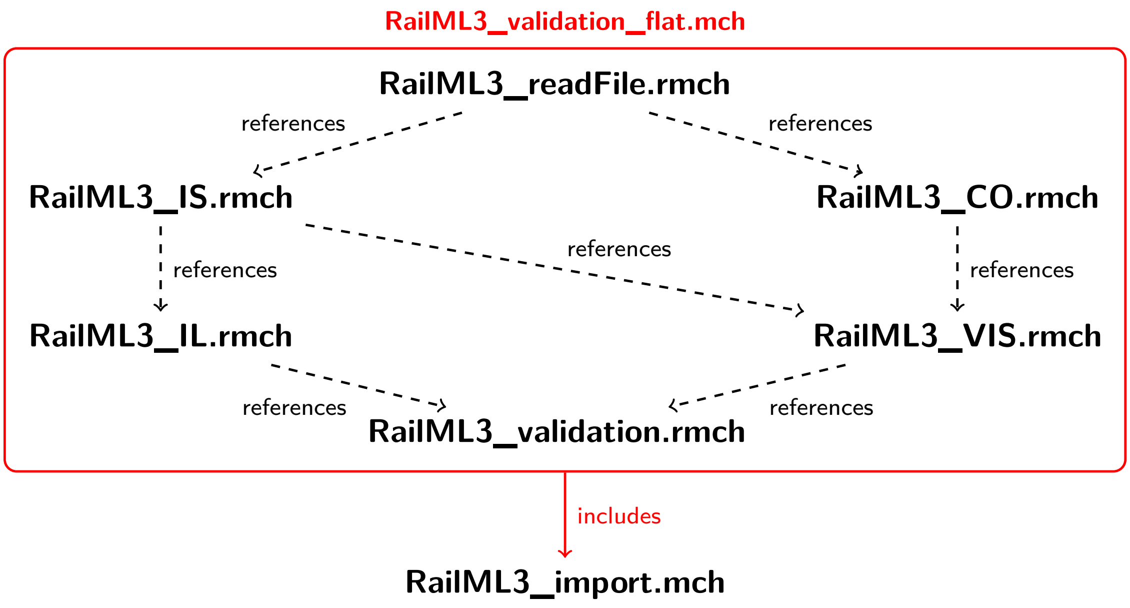


Figure 2: Machine Hierarchy of the Import and Validation Process

Furthermore, all IDs occurring in the XML document are extracted into the concrete constant `all_ids` of type `STRING <-> (STRING <-> NATURAL)`, where a tuple of this relation contains the ID (`attributes("id")`) as the first argument and a pair of its type (`element`) and record number (`recId`) as the second argument. The record number comes from the sequence given by `READ_XML` and ensures uniqueness of the pair. As can be seen, all IDs are treated as strings, which in some places makes it easier to combine IDs of different types. The rule `unique_ids` is then used to check whether all IDs found in the document are unique, which is crucial for the correct functionality of the model, as many elements may reference others[3]. Finally, for easy access to all IDs of a certain type there is the abstract constant `allIdsOfType` of type `STRING +-> FIN(STRING)`, which returns the set of all IDs for a given type.

There are also two other useful abstract constants for the import, namely `elementsOfType` and `childsOfElementType`. `elementsOfType` returns all records of `data` of the given type, and `childsOfElementType` does the same, except that it returns only those elements that are children of a parent specified by its `pId`.

For efficient use of these abstract constants, the `MEMOIZE_FUNCTIONS` preference should be enabled in ProB (cf. Section 11.3).

[3]See also `https://wiki3.railml.org/wiki/Dev:identities` (accessed on 26/10/2023).

3.2 Data Conversion

Once the XML file has been read, the data is converted and validated for use in the B-model by the beforehand mentioned schema-based rules machines. Conversion to standardised data types is necessary to ensure correct interpretation of the data values [St-23]. For example, numerical values without conversion to the corresponding B-type (`INTEGER` or `REAL`) would only be interpretable as a string. This would not allow any calculations with the values and would also lead to further inconsistencies like `"50.0"` not being equal to `"50.00"`, even though both strings represent the same number.

Currently, only selected elements and attributes are imported and covered by the validation, mainly those that are important for the animation, but also additional ones. The general approach of the validation is to first store data of the same type in a set of B-records, which have fields for the attributes and also for certain children. Children are stored in the parent record if their type is uniquely associated with the type of the parent, such as `overlapRelease` belongs to `overlap`, or if they can be formulated as simple sets, for example, as a set of referenced IDs. The types of these records are defined in separate definition files, such as `RailML3_IS_Types.def`. Since a type must be specified for each field of a record, the types of the imported data are automatically validated by ProB in the form of invariants, as is the case for all computations of rules machines. In the case of an error, there is an invariant violation that must be investigated by the user.

An example definition of a record type for railML elements of the type `tvdSection` can be found in Listing 12, together with its corresponding computation in Listing 13, which performs the conversion from strings to the appropriate types. As for all record types that are not integrated into the parent record, the first two elements contain the record and the parent ID given by `READ_XML` to keep the relationships between the XML elements. The `xmlLineNumber` is also given as meta information by `READ_XML` and can be easily converted to a natural number using `STRING_TO_INT`. This field is mainly used to provide line positions in case of a validation error.

Listing 12: B Record Type Definition for `tvdSection`

```
RailML3_IL_tvdSection_Type ==
  struct(
    recId : NATURAL1,
    pId : NATURAL,
    xmlLineNumber : NATURAL1,
    // attributes:
    Id : FIN(dom(all_ids)),
    isBerthingTrack : BOOL,
    technology : FIN(RailML3_IL_TVD_SECTION_TECHNOLOGY_TYPES),
    // children:
    hasDemarcatingBufferstops : FIN(allIdsOfType("bufferStop") \/
        RailML3_IS_OPENEND_IDS),
    hasDemarcatingTraindetectors : FIN(allIdsOfType("trainDetectionElement"
        ))
  );
```

Listing 13: Computation for Import of `tvdSection`

```
COMPUTATION set_tvdSection
DEPENDS_ON_RULE is_valid_tvdSections
DEPENDS_ON_COMPUTATION set_BORDER
BODY
  DEFINE RailML3_tvdSection
  TYPE FIN(RailML3_IL_tvdSection_Type)
  VALUE dom({e, e_tvd | e_tvd : elementsOfType("tvdSection")
    & e = rec(
      recId: e_tvd'recId,
      pId: e_tvd'pId,
      xmlLineNumber: STRING_TO_INT(e_tvd'meta("xmlLineNumber")),
      Id: e_tvd'attributes[{"id"}],
      isBerthingTrack: IF "isBerthingTrack" : dom(e_tvd'attributes) & e_tvd
          'attributes("isBerthingTrack") = "true" THEN TRUE ELSE FALSE END,
      technology: IF "technology" : dom(e_tvd'attributes) THEN
          {TYPED_STRING_TO_ENUM(RailML3_IL_TVD_SECTION_TECHNOLOGY_TYPES,
          "technology_"^e_tvd'attributes("technology"))} ELSE {} END,
      hasDemarcatingBufferstops: dom({ i_A, e_A | e_A :
          childsOfElementType("hasDemarcatingBufferstop", e_tvd'recId)
          & i_A : e_A'attributes[{"ref"}] }),
      hasDemarcatingTraindetectors: dom({ i_A, e_A | e_A :
          childsOfElementType("hasDemarcatingTraindetector", e_tvd'recId)
          & i_A : e_A'attributes[{"ref"}] })
    ) })
  END
END;
```

The handling of attributes is a bit more challenging, since they do not always have to be present in XML files. There can be optional attributes that are correctly absent, but there are also obligatory attributes that can be incorrectly absent. To handle the absence of an attribute, it is therefore necessary to treat attributes as sets of their type, which allows an unspecified attribute to be represented by the empty set. This can be observed, for example, in Listing 12 with its field `Id`, even though this is an obligatory attribute. The presence of such attributes will be checked in the syntactic validation afterwards. Attributes that must be defined for correct animation and for which default values can be applied are not modelled as sets. Their value is either selected as specified in the railML file or, if not specified, the default value is used. In the example this is the case for the attribute `isBerthingTrack` which is set to `FALSE` if it is not specified.

Some fields may only contain predefined values, as it is the case with the `technology` attribute. These values can be obtained from the documentation of each element in the railML wiki[4] and are modelled in B as enumerated sets like

```
RailML3_IL_TVD_SECTION_TECHNOLOGY_TYPES =
        {technology_axleCounter, technology_trackCircuit}.
```

[4]For `tvdSection`: https://wiki3.railml.org/wiki/IL:tvdSection (accessed on 26/10/2023).

As can be seen in line 14 of Listing 13, the conversion is done by the external function `TYPED_STRING_TO_ENUM`, which takes as arguments the expected enumerated set and the string to be converted, and throws an error if the conversion is not possible, meaning that an invalid entry is used in the railML file.

For elements that specify times in the format *xs:duration*, this can be translated to a natural number representing the duration in milliseconds using the definition `readDuration`, which uses regular expressions for parsing. At the moment, only seconds and milliseconds are supported.

In general, the described procedure is applied for child elements that are integrated into the parent node, provided that multiple attributes and additional children are used. The child elements of an element of a particular type are obtained using the abstract constant `childsOfElementType` as described in Section 3.1. If the children only specify references to other elements, such as the demarcating elements of `tvdSections`, they are combined into one set of references, as shown in lines 15 and 16 of Listing 13. The type of the field and the set of IDs can be used to specify which references should be allowed, here these are for example `bufferStops` and open ends (`borders` with attribute `openEnd="true"`) for `hasDemarcatingBufferstops` (line 11 in Listing 12).

3.3 Syntactic Validation

After conversion of the elements of a type, a rule `is_valid_<type>` validates whether all elements of the previously computed record `RailML3_<type>` are syntactically valid under the railML 3 schema corresponding to the version of the file (3.1 or 3.2). Of course, this task can also be performed by other validation software, such as *railVIVID*[5], but it is essential to check this in the B model as well for reasons of well-definedness. The example of `tvdSections` is continued in Listing 14 with three examples of frequently occurring rules.

Obligatory Attributes The first rule (lines 4-8 in Listing 14) checks whether the `Id` field is filled with a valid value as it is an obligatory attribute. In addition, the invariant generated by the computation checks whether the type of the ID matches the type determined when the file was read (which should always be the case). There is no need to check for duplicate IDs here, as this is already done generically when the file is read. The purpose of this check is only to ensure that each element of the type has a correct `Id` attribute. This is important for the well-definedness of subsequent rules that use this type, and allows the use of `MU_WD`[6] to access the `Id` field.

[5] `https://www.railml.org/en/user/railvivid.html` (accessed on 26/10/2023)

[6] The `MU_WD` operator is available in ProB since version 1.12.2 and gives the element of a singleton set, which is always the case for a validated obligatory attribute. Its semantic is similar to `MU`, but `MU_WD` provides stronger propagation if the used expression is known to be well-defined.

Listing 14: Syntactic Validation Rule for `tvdSection`

```
RULE is_valid_tvdSection
DEPENDS_ON_COMPUTATION set_tvdSection
BODY // [...]
    RULE_FAIL e
        WHEN e : RailML3_tvdSection & (card(e'Id) /= 1 or (card(e'Id) = 1 &
             MU_WD(e'Id) = ""))
    COUNTEREXAMPLE
        errorAttribute("id", "tvdSection", e'xmlLineNumber)
    END;
    RULE_FAIL e_tvd, c
        WHEN e_tvd : RailML3_tvdSection & c = card(e_tvd'
           hasDemarcatingBufferstops) & c /: 0..2
    COUNTEREXAMPLE
        errorCard("hasDemarcatingBufferstop", {e_tvd}, 0, 2, c)
    END; // [...]
    RULE_FORALL e
        WHERE e : RailML3_tvdSection & card(RailML3_tvdSections) = 1
        EXPECT e'pId = MU_WD(RailML3_tvdSections)'recId
    COUNTEREXAMPLE
        errorParent("tvdSection", "tvdSections", e'xmlLineNumber)
    END
END;
```

An example of frequent use of `MU` is the polymorphic definition `elementsOfId`, which takes as first argument an arbitrary set of records of converted and validated railML data (of any type, as long as it contains a field `Id`), and as its second argument an ID string to obtain the record of the corresponding ID:

```
elementOfId(Set, eId) == MU({ e | e :  Set & eId = MU(e'Id)}).
```

Any other obligatory attributes are validated in the same way. There are certain attributes that are not obligatory by the railML schema, but necessary for a correct setup of the formal model. These are made obligatory by validating their existence like its done for obligatory attributes or by using default values if they are not specified and a reasonable value exists (see below).

References Obligatory attributes that require additional validation are `ref` attributes for referencing other elements, since treating IDs as strings allows arbitrary references. Incorrect references may cause well-definedness errors later in the B model, which makes an additional type check by rules necessary in addition to the invariant check. An attempt was made to avoid this by aborting the import process early on an invariant violation using the values provided by `FORMULA_VALUES("inv")`, but this did not work because the invariants are evaluated after the guards when computing a new state. Therefore, not well-defined rules would still be able to be executed at least for one step, necessitating the additional validation. However, this step is generally not necessary for the remaining obligatory attributes, as their values have already been converted to their specific type.

Children Cardinalities Besides presence of obligatory attributes, an important property to validate is the number of children of an element of a certain type. For example, an element of type `route` is expected to have exactly one child of type `routeEntry`. Otherwise the route cannot be correctly inferred and should not be accepted as a valid route during syntactic validation. Some elements have a rather loose requirement for children cardinalities given by the railML schema, which is not strong enough to guarantee a correct animation in B. Such elements are, for example, switches, where a switch of type `ordinarySwitch` is required to have exactly two branches (left and right) for correct animation, whereas the railML schema only requires it to have zero to two branches. The syntactic validation of these elements is done in the sense of a stricter interpretation of the schema. An example of a rule validating children cardinalities can be found in Listing 14 (lines 9-13). The check itself uses the set of children of a given type of a corresponding parent element and compares the cardinality with the requirements of the schema. If there is a mismatch, the rule fails.

As a general check, it is also ensured that basic essential elements are present. This concerns for example the `infrastructure` and `topology`, but also the sections `netElements` and `netRelations` (without which no topology can be built).

Values Rare cases require a check of the input value provided in the railML file. These are for example intrinsic coordinates which must be within the range between 0.0 and 1.0. A reasonable interpretation of other values is impossible, so they are rejected to avoid an incorrect behaviour of the B model.

Default Values Attributes that are not specified in the railML file, but that are required for correct animation in B are provided with default values where possible. For some attributes, the schema itself provides default values, otherwise plausible values have been chosen, which are documented in Table 7 in Appendix C. In some cases, values can also be constructed from others, such as intrinsic coordinates, if the length of the net element and a position are given (cf. Section 4.1).

Parent Types Finally, for each element is checked whether the parent of an element is of a valid type for that element. This is illustrated by the last part of the rule in Listing 14 (lines 14-19). In this example, it is checked that each element of type `tvdSection` has the parent of type `tvdSections`. This rule can succeed if there is no `tvdSections` element at all. In this case, the error is covered by checking for correct cardinalities of the children (`tvdSection` must not exist without `tvdSections`, i.e. no `tvdSection` is expected).

Counterexamples If one of these validations is violated for an element, a counterexample is provided by the rule specified in the B-Rules DSL. For the affected elements, the line numbers are provided as additional information in the textual user feedback. Most of the counterexample messages can be reused and are thus separated into definitions. One of these is the definition for validation errors regarding parent types shown in Listing 15:

Listing 15: Definition of a Validation Error Message

```
errorParent(Type, ParentType, lineNr) ==
    STRING_FORMAT("[Line ~w]: expected parent of type '"^ParentType
       ^"' for type '"^Type^"'", lineNr).
```

The other definitions of counterexamples are structured in the same way.

3.4 Derive Relations from the Data

After successful syntactic validation of a `<type>`, a corresponding subsequent computation `set_<TYPE>` becomes enabled, where sets and relations are computed for easy access to the data by the animation machine and also for semantic validation. These computations use the previously validated records and establishes relationships such as the path of a route, which can then be validated semantically. Furthermore, some fields are made accessible more convenient for use in the animation machine. An example is the function `RailML3_IL_TVD_SECTION_BERTHING_TRACKS` in Listing 16, which maps IDs of `tvdSections` to the value of their associated attribute `isBerthingTrack`. As a side note, this also demonstrates how to conveniently obtain the value of an attribute for an imported element using `elementOfId` as introduced in the previous Section 3.3.

Listing 16: Computation of Relations for Imported `tvdSections`

```
COMPUTATION set_TVD_SECTIONS
DEPENDS_ON_RULE is_valid_tvdSection
DEPENDS_ON_COMPUTATION set_NET_ELEMENT, set_NET_RELATION,
    set_NET_RELATION_SUBSEQUENT_BLOCKS, set_SPOT_LOCATION
BODY
    DEFINE RailML3_IL_TVD_SECTIONS // [...]
    DEFINE RailML3_IL_TVD_SECTION_BERTHING_TRACKS
        TYPE allIdsOfType("tvdSection") --> BOOL
        VALUE %i_tvd.(i_tvd : allIdsOfType("tvdSection") |
            elementOfId(RailML3_tvdSection, i_tvd)'isBerthingTrack )
    END;
    DEFINE RailML3_IL_TVD_SECTION_DEMARCATING_ELEMENTS // [...]
END
```

Listing 16 shows the typical structure of such a computation. It depends on the exemplary validation rule from Listing 14 and any other already derived relations needed for this computation. As stated in Section 2, the B-Rules DSL should generally capture such dependencies implicitly. Despite this, the dependencies are listed explicitly to ensure proper functioning and enhance comprehensibility.

Furthermore, some computations capture properties by their type definition (e.g. when a mapping of interlocking IDs to corresponding infrastructure IDs is defined as a bijective function, this forces each `IS` element to have exactly one related `IL` element), which is handled and checked as an invariant by the B-Rules DSL.

More details regarding the resulting data structures and modelling principles are explained for selected types in Section 4.

3.5 Semantic Validation

The import process of an element type is concluded by its semantic validation. This phase of the import is possibly the most interesting for the application of the B-method, as here the relationships and properties of the specified elements can be easily checked with the formalisms of the B-method. For this purpose, there is the subsequent rule `validate_<type>`, which validates not on the validity of the schema, but on mandatory semantic properties. This includes both the properties of data computed in the previous step, such as whether a route completely connects the entry and exit signal, as well as those of directly imported elements from Section 3.2, such as switch branches that should intersect.

The rule is optionally followed by a rule called `warnings_<type>`, which issues warnings if a non-mandatory property is not met, but the information may still be of interest to the user, for example if two overlapping routes have been specified. Additionally, warnings are generated if a default value has been set for an attribute (Section 3.3). To distinguish between the two types of warnings, the B-Rules DSL error types are used. Semantic warnings are generated with standard error type 1, while warnings related to default values are generated with error type 2.

As a source for rules, a few semantic constraints are available in the railML 3 Wiki[7], but they do not refer to infrastructure and interlocking data, so the constraints validated here are mostly based on own considerations and can be adapted easily as soon as more constraints become available. Other rules, some of which are implemented in the rules, can also be found in railOscope[8]. The implemented validations and warning rules are all listed in Table 8 in Appendix C for clarity.

The structure of the rules is quite similar to those of syntactic validation. The counterexamples of the validations and warnings are provided as in Section 3.3 with individual messages containing line number, ID, and additional information on the violation.

[7]`https://wiki3.railml.org/wiki/Dev:Semantic_Constraints` (accessed on 26/10/2023)

[8]`https://railoscope.cloud.xwiki.com/xwiki/bin/view/Main/TopoEditor/Validations` (accessed on 6/9/2023)

3.6 Use of the Imported Data

The process described in Sections 3.1 to 3.5 is repeated for all required element types. In doing so, the dependencies to other element types, which must already be validated for the import of a certain type, must be specified in the headers of the rules and computations for this type. Individual rules can be specified in the `RailML3_validation.rmch` machine if desired. However, since it is currently not possible to include rules machines in ordinary machines, changing these rules is only possible if the internal machine is re-exported afterwards, which is not user-friendly. For custom rules it is instead recommended to use the generated validation machine as described in Section 6.2.

Once the rules machines have been configured, the internal representation of the topmost machine (`RailML3_validation.rmch`) must be stored as a separate machine (`RailML3_validation_flat.mch`). Additionally, it is necessary to change the machine name from `__RULES_MACHINE_Main` to `RailML3_validation_flat` and to remove the line binding the constant `file` to a filename. As highlighted in Figure 2, the internal representation of `RailML3_validation.rmch` can be considered as a package containing all rules for railML import and validation, which can be accessed via the machine `RailML3_import` as interface. Clearly, the operations could also be performed directly in the rules machines. However, this then only provides a pure validation and does not allow any further use of the data in ordinary B machines. In order to make use of the imported data in any machine, it is sufficient to include the `RailML3_import` machine (provided that the `RailML3_validation_flat.mch` is also available in the same directory) as outlined in Figure 3, add the operation `importRailML` to the machine's `INITIALISATION`, and specify the railML file to be loaded by setting the constant `file` to its path. The import machine can remain unchanged as long as no new rules or computations are added. When adding new operations, the order of the dependencies must be respected.

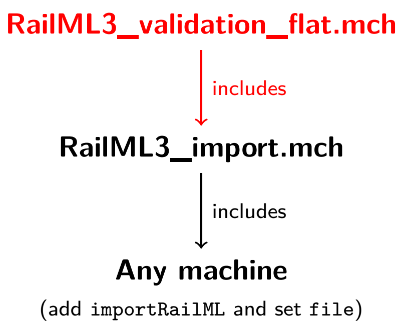


Figure 3: Generic Machine Hierarchy for Use of the Imported Data

The main task of the import machine is done by its operation `importRailML`, which executes all rules and computations in a valid order that respects the dependencies and outputs possible errors and warnings to the user. For this purpose, the machine contains three definitions, namely `runErrorRule`, `runWarningRule`, and `runComputation`. For a full import, the import operation must be called with the parameter `TRUE`. If called with `FALSE`, the import will stop when all the data required for visualisation has been processed. Additionally, a boolean variable `no_error` keeps track of whether a validation error has occurred during the import. Rules and computations are only executed if `no_error` is `TRUE`, otherwise all subsequent operations after the failing rule are skipped and the import is aborted. `no_error` is not affected by occurrence of warnings. The definition for execution of a validation rule (whose failure is an error) is shown in Listing 17.

Listing 17: Definition for Execution of a Validation Rule in the Import Machine

```
runErrorRule(Rule) ==
    IF no_error = TRUE THEN
        VAR Var1, Var2 IN
            Var1, Var2 <-- Rule;
            IF Var1 = "FAIL" THEN
                no_error := FALSE;
                ADD_ERRORS(Var2[{1}]);
                ADD_STATE_ERRORS(Var2[{1}]);
                ADD_ERROR("RailML: ", "File import aborted, the
                    specified file does not contain valid railML.")
            ELSE skip END
        END
    ELSE skip END;
```

The passed rule is executed if no error has occurred previously, and the first return value `Var1` is used to check if the rule was successful. If not, an error is raised using the external function `ADD_ERROR` and a state error is created for each provided counterexample in `Var2` using `ADD_STATE_ERRORS`. The latter was added to ProB in version 1.12.3 in the context of this project, as was `ADD_WARNINGS` for creation of warnings. These external substitutions expect a set of strings which ProB will then output to the user as individual errors or warnings. Warnings that occur during the execution of a warning rule with `runWarningRule` are stored in the sets `semanticWarnings` or `defaultWarnings`, depending on the type of warning. At the end, each of the sets is output together using `ADD_WARNINGS`.

During development, it was also considered to distinguish warnings and errors by making use of the error types of the B-Rules DSL (as now only for warnings) instead of using two different rules for each. However, this approach did not lead to success, as it could not be distinguished whether a rule failed only because of a warning or because of an error. This is because the predicates provided for this in the B-Rules DSL cannot be used in the statically evaluated `ACTIVATION` predicate of a rule (Section 2). In the end, this means that if a rule fails with a warning, the import would get stuck because the subsequent rules remain disabled.

Finally, the computations are executed by the definition `runComputation`, which simply executes the passed operation if no error occurred before, and skips it otherwise.

4 Modelling of the Data in B

During the import, additional relations and sets are computed (Section 3.4) to model the relationships between the different elements. This chapter addresses important decisions and issues of modelling these in the formal B-method. In general, it should be noted that the imported and validated data in its current state represents only a subset of the very detailed railML 3 schema. The current selection of imported elements focuses on topology, animatable infrastructure elements and key elements for modelling interlocking logic. An overview of all derived data structures can be found in Table 6 in Appendix C together with a brief description of their characteristics.

A common problem with formal modelling of non-formal schemas is that they may be interpreted differently by users, leading to inconsistencies in usage that the formal model has to deal with [St-23]. Another related problem is the lack of data, which is essential for the correct functioning of the model. To overcome this, there are several points during the import process where attempts are made to infer the missing data from other given data, such as intrinsic coordinates for locations, track ends, activation blocks for routes, and signal plans. A selection of these are presented in this section.

4.1 Representation of Locations

A crucial concept is the modelling of locations and their connectivity. Abrial [Abr10] defines blocks in his train system model in terms of an abstract element that can be connected to further blocks, but without the possibility of specifying the location of a train within a block. Gruteser et al. [Gru+23] use an extended model, which uses discrete values for locations of objects within a block without specifying a direction. These approaches are not sufficient for modelling railML data. In railML 3, the locations for all infrastructure elements are specified in terms of `netElements` and intrinsic coordinates within them. It is therefore important for the correctness of the model's behaviour to be able to capture the exact train position within a net element.

Locations Each location is represented by a triple of the type

`STRING * REAL * RailML3_IS_DIRECTION`,

containing the ID of the `netElement`, the intrinsic coordinate within the net element and the application direction of the location (only *normal* and *reverse*). This approach is also proposed by Martins et al. [Mar+22], who translate railML data into a rule language inspired by linear temporal logic and Alloy. The use of real numbers for the intrinsic coordinates is possible due to a (currently still experimental) extension of classical B by the `REAL` type in ProB [Rut23]. This allows the decimal values to be imported directly from the XML file using the new external function `STRING_TO_REAL`, without having to discretise them as would otherwise be necessary in classical B.

As described in Section 1.2, the locations of infrastructure elements can be either specified by spot or by linear locations, which are imported as generic child elements for all elements of the entire file, regardless of whether the associated type is also imported into B. The spot locations can be accessed via the function `RailML3_IS_SPOT_LOCATIONS`, which is of type `STRING +-> FIN(STRING * REAL * RailML3_IS_DIRECTION)` and maps the ID of an (infrastructure) element to its spot locations. These have to be modelled as a set, since the schema in principle allows any number of spot locations for an element. The approach for linear locations is almost the same as for spot locations, except that these are relations connecting the associated net elements at the given intrinsic coordinates. The start and end coordinates are connected by the net relations, which are stored in `RailML3_IS_NET_RELATION`, and then kept in `RailML3_IS_LINEAR_LOCATION_ASSOCIATED_NET_ELEMENTS`. To take into account the direction of application, the relation is stored either as such for the direction *normal*, inverted for *reverse*, or both for the direction *both* in `RailML3_IS_LINEAR_LOCATIONS`. Also, in case of multiple linear locations for the same element, all locations are merged into one relation containing all linear locations of the referenced element, as there would be no gain in storing them as individual relations and no information is lost.

Relation Containing All Locations Since the intrinsic coordinates are treated as real numbers, it is obvious that not all locations within a net element can be modelled. The locations of interest for modelling are precisely those on which infrastructure elements are located. These are contained in the spot and linear locations. All these locations together with the endpoints of each net element (each for both normal and reverse direction) are considered as part of the relation to be computed. For each net element, the locations of infrastructure elements are sorted by their intrinsic coordinate and integrated into an inner relation connecting both endpoints according the determined coordinate order. This is done by the computation `set_NET_RELATION_SUBSEQUENT_LOCATIONS`, which yields the relation `RailML3_IS_NET_RELATION_SUBSEQUENT_LOCATIONS` connecting the inner relations at their endpoints with respect to the specified `netRelations` on the microscopic level (Section 1.2). This "topology relation" contains the complete topology together with the locations of the infrastructure elements, enabling precise modelling of train movements on the net elements, but also the computation of paths between locations, e.g. for routes.

To allow access to information about the connectivity of two locations and also for performance reasons, the transitive closure of the relation is precomputed and stored in `railML3_IS_NET_RELATION_SUBSEQUENT_LOCATIONS_closure1`. To simplify later computations of route, overlap, and TVD section paths, it is assumed that the topology does not contain cycles (cf. `ENV-11` [Abr10]). To achieve correct results for pairs of locations, it is important to specify the directions correctly. If, for example,

`("ne1", 0.25, direction_normal)` $\mapsto$ `("ne2", 0.0, direction_reverse)`

is a pair connected by the transitive closure, the following would not be connected in it:

`("ne1", 0.25, direction_normal)` $\mapsto$ `("ne2", 0.0, direction_normal)`.

The directions are induced by the net relations: if a net relation connects two endpoints with the same intrinsic coordinates, this indicates a change of direction (e.g. $1.0 \to 1.0$). Connecting different intrinsic coordinates, such as $1.0 \to 0.0$, keeps the direction (here *normal*). Within a net element, the direction *normal* indicates that the inner relation is traversed from 0.0 to 1.0, for *reverse* the same applies in the opposite direction.

Originally, the computation of the subsequent locations within the topology was performed by a recursive `FUNCTION` (B-Rules DSL) without using directions for locations. Instead a `FOR`-loop was used, which traverses the net relations from a given location up to the next change of direction and then recursively continues the search for the opposite direction. Although this approach also gives a correct relation, it leads to both performance problems and a significantly higher implementation effort, as both directions have to be considered when starting search at a location. Therefore, the directions were included in the locations, which greatly simplified the computations, especially of the transitive closure.

Example

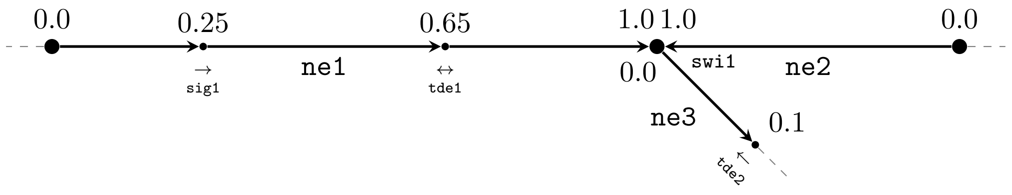


Figure 4: Concept of Subsequent Locations for the Relational Representation

Figure 4 shows an exemplary topology together with a few locations of infrastructure elements. These are the signal `sig1` located at `ne1` which operates only in normal direction, the switch `swi1` located at the intersection of `ne1`, `ne2` and `ne3` and the two train detection elements `tde1` and `tde2`. This results in the following set of spot locations `RailML3_IS_SPOT_LOCATIONS`:

$$\begin{aligned}\{&\texttt{"sig1"} \mapsto \{(\texttt{"ne1"},0.25,\texttt{direction\_normal})\},\\ &\texttt{"swi1"} \mapsto \{(\texttt{"ne1"},1.0,\texttt{direction\_normal})\},\\ &\texttt{"tde1"} \mapsto \{(\texttt{"ne1"},0.65,\texttt{direction\_normal}),(\texttt{"ne1"},0.65,\texttt{direction\_reverse})\},\\ &\texttt{"tde2"} \mapsto \{(\texttt{"ne3"},0.1,\texttt{direction\_reverse})\}\}.\end{aligned}$$

Note that the location of `tde1`, which is applicable in *both* directions, is split into a *normal* and a *reverse* directed location in order to be able to apply them as locations to the relation. A possible linear location could lead from (`ne1`, 0.25) to (`ne2`, 0.0) with `application-Direction="reverse"`. This gives `RailML3_IS_LINEAR_LOCATIONS` as follows:

$$\begin{aligned}\{\texttt{"trk1"} \mapsto \{&(\texttt{"ne2"},0.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne2"},1.0,\texttt{direction\_normal}),\\ &(\texttt{"ne2"},1.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne1"},1.0,\texttt{direction\_reverse}),\\ &(\texttt{"ne1"},1.0,\texttt{direction\_reverse}) \mapsto (\texttt{"ne1"},0.25,\texttt{direction\_reverse})\}\}.\end{aligned}$$

Observe that linear locations contain only the start and end locations connected by net relations without the inner locations (e.g. (`ne1`, 0.65)). The complete topology relation containing all locations would be in this example:

$$
\begin{aligned}
&\{(\texttt{"ne1"},0.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne1"},0.25,\texttt{direction\_normal}),\\
&(\texttt{"ne1"},0.25,\texttt{direction\_normal}) \mapsto (\texttt{"ne1"},0.65,\texttt{direction\_normal}),\\
&(\texttt{"ne1"},0.65,\texttt{direction\_normal}) \mapsto (\texttt{"ne1"},1.0,\texttt{direction\_normal}),\\
&(\texttt{"ne1"},1.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne2"},1.0,\texttt{direction\_reverse})\},\\
&(\texttt{"ne1"},1.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne3"},0.0,\texttt{direction\_normal})\},\\
&(\texttt{"ne2"},1.0,\texttt{direction\_reverse}) \mapsto (\texttt{"ne2"},0.0,\texttt{direction\_reverse})\},\\
&(\texttt{"ne3"},0.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne3"},0.1,\texttt{direction\_normal})\},\\
&(\texttt{"ne1"},1.0,\texttt{direction\_reverse}) \mapsto (\texttt{"ne1"},0.65,\texttt{direction\_reverse}),\\
&(\texttt{"ne1"},0.65,\texttt{direction\_reverse}) \mapsto (\texttt{"ne1"},0.25,\texttt{direction\_reverse}),\\
&(\texttt{"ne1"},0.25,\texttt{direction\_reverse}) \mapsto (\texttt{"ne1"},0.0,\texttt{direction\_reverse})\},\\
&(\texttt{"ne2"},1.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne1"},1.0,\texttt{direction\_reverse})\},\\
&(\texttt{"ne2"},0.0,\texttt{direction\_normal}) \mapsto (\texttt{"ne2"},1.0,\texttt{direction\_normal})\},\\
&(\texttt{"ne3"},0.0,\texttt{direction\_reverse}) \mapsto (\texttt{"ne1"},1.0,\texttt{direction\_reverse})\},\\
&(\texttt{"ne3"},0.1,\texttt{direction\_reverse}) \mapsto (\texttt{"ne3"},0.0,\texttt{direction\_reverse})\}.
\end{aligned}
$$

Derivation of Intrinsic Coordinates As can be observed, the intrinsic coordinates are mandatory for specifying locations in B. Hence, another crucial aspect is the derivation of intrinsic coordinates for locations whose intrinsic coordinates are not explicitly given. If a valid attribute `intrinsicCoord` is provided for a `spotLocation` or `intrinsicCoordBegin-/End` for `associatedNetElements` of a `linearLocation`, they can be used as such and nothing needs to be derived. If none of these is specified, it is tried to derive the intrinsic coordinate from the `pos` or the `posBegin-/End` attribute in combination with the length of the associated `netElement`:

$$\texttt{intrinsicCoord}_{derived} = \frac{\texttt{pos}}{\texttt{length}_{@netElementRef}}.$$

Clearly, this approach requires both attributes to be specified. If none of these approaches can be applied, the `intrinsicCoord` for `spotLocations` is set to 0.0 as the fallback value. For associated net elements of linear locations, the in railML 3.2 obligatory `keepsOrientation` attribute provides basic information about the orientation of the net element. Using this, the fallback values here are chosen as $\texttt{intrinsicCoordBegin} = 0.0$ and $\texttt{intrinsicCoordEnd} = 1.0$ for $\texttt{keepsOrientation} = \texttt{TRUE}$ and vice versa for `FALSE`. However, the `keepsOrientation` attribute has become obsolete with railML 3.2 and will be removed in future releases. The related discussion[9] has shown that localisation in railML 3 should be more restricted by additional rules to avoid ambiguity. The outcome of the discussion will be monitored and the procedure described here will be adapted if necessary.

[9] https://www.railml.org/forum/index.php?t=msg&th=818 (accessed on 19/9/2023)

According to the railML 3 schema, the attributes of intrinsic coordinates are optional for the location elements, and furthermore both spot and linear locations are themselves optional. This means that for some elements there may be no location or more than one location, even of the three different types. As this may be too little information or otherwise lead to ambiguity on the part of a railML import interface, this issue is currently part of the discussions for the future development of railML 3[10]. As mentioned in the linked post, it is planned to reduce the ambiguity by a ruling framework that can easily be added to the semantic validation rules of Section 3.5 once it has been added as a semantic constraint to the schema. The current semantic validation rules formulated in Section 3.5 already force certain types of infrastructure elements to have exactly `spotLocation` or `linearLocation` to avoid this problem. However, this also does not fully comply with the schema and can lead to problems if there is more than one `linearPositioningSystem`. Therefore, these rules only serve as a temporary solution until the railML community has agreed on an approach.

4.2 Representation of Movable Elements

Movable elements that can be specified in railML 3 are mainly switches, single and double switch crossings, movable ordinary crossings, and derailers[11]. As already described in Section 1.2, each movable element has an infrastructure and an interlocking object in railML 3. For this to work correctly for the movable elements, there must be at least one matching infrastructure object for each movable interlocking element, which is therefore checked by the validation rules of the rules machines. In general, each movable element can be in an undefined position or in one of the predefined interlocking positions. These positions are, extending the assumptions `ENV-1`/`ENV-2` [Abr10]:

Switch:	`left`, `right`
Derailer:	`derailingPosition`, `passablePosition`
Movable Crossing:	`downleft-rightup`, `upleft-rightdown`

As multiple interlocking elements can refer to the same infrastructure element, one of these positions is not necessarily enough to infer the position on infrastructure side. For the B model, it is assumed that crossings and derailers have exactly one associated interlocking element as multiple of these elements would introduce ambiguity. This includes the assumption that two-sided derailers are modelled as one synchronously switching derailer. In contrast, switches can have several interlocking elements to model switch crossings. This means that for switches, the possible states on the infrastructure side are different from those on the interlocking side:

[10] `https://www.railml.org/forum/index.php?t=msg&th=920` (accessed on 19/9/2023)
[11] See also `https://wiki3.railml.org/wiki/Dev:Moveable_Elements` (accessed on 27/10/2023).

Ordinary Switch:	`left`, `right`
Single Switch Crossing:	*upleft-rightdown*, *downleft-rightup*,
one of:	*upleft-rightup*, *downleft-rightdown*
Double Switch Crossing:	*upleft-rightdown*, *downleft-rightup*,
	upleft-rightup, *downleft-rightdown*.

Note that these positions are not modelled as such in B. They are described by a combination of positions of the interlocking switches by the function `RailML3_IL_SWITCH_BRANCHES`, whose domain is of the type

```
RailML3_IS_SWITCH_IDS -->
        (allIdsOfType("switchIL") +-> RailML3_IL_SWITCH_POSITIONS).
```

It contains, for each infrastructure switch element[12], all combinations of possible positions of the associated interlocking switch elements and the resulting branch on track side. Additionally, the variable `RailML3_IS_SWITCH_BRANCHES` provides more generic information about the position of a switch within the topology, by containing all possible traversing paths over a movable element given by its location and branches specified in the infrastructure schema.

The implementation for movable crossings is designed in the same way by `RailML3_IS_CROSSING_BRANCHES` and `RailML3_IL_MOVABLE_CROSSING_BRANCHES`, except that the domain of the latter expects only one pair of the uniquely associated interlocking element with its position instead of a partial function. Also not all crossings are taken into account, but only those having a related interlocking object for movability information. The remaining ones are assumed to be static, non-movable crossings.

Finally, for derailers, the concept of branches does not apply, but the location is computed in `RailML3_IS_DERAILER_NOT_PASSABLE` in terms of non-passable pairs analogous to the infrastructure branches computed above.

There are also additional characteristics like related movable elements for synchronous or dependent switching of multiple movable elements in combination with position restrictions implemented. See Table 6 in Appendix C for a brief description of the corresponding data structures. Further explanations on the concept of crossings, derailers and switches in railML 3 are available in the railML wiki[13].

[12]`RailML3_IS_SWITCH_IDS` excludes the IDs of `switchIS` elements of type `switchCrossingPart` from `allIdsOfType("switchIS")`, as the parent element of a switch crossing is sufficient for correct modelling and its parts cannot have independent behaviour.

[13]`https://wiki3.railml.org/wiki/IL:{derailerIL,movableCrossing,switchIL}` (accessed on 23/9/2023)

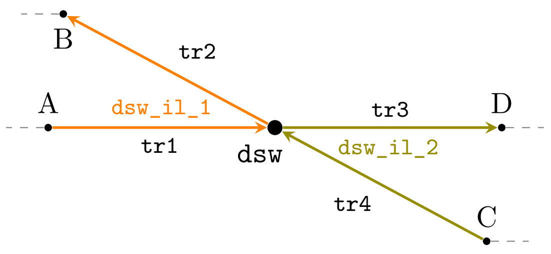


Figure 5: Concept of a Double Switch Crossing

Example Figure 5 illustrates an example that defines a double switch crossing `dsw` on infrastructure side and its two associated interlocking switches `dsw_il_1` and `dsw_il_2`. Both can take either the position `left` or `right`. Note that for the interlocking representation of a switch, the branches are indicated in terms of tracks (`tr1`-`tr4`). In this case, `dsw_il_1` corresponds to the branch definition

`<branchLeft ref="tr1"/> <branchRight ref="tr2"/>`.

The same applies to `dsw_il_2` with `tr3` and `tr4`. Combined, this gives the following encoding for the branch leading from B to D, which corresponds to the implicit position *upleft-rightup*:

$$\texttt{"dsw"} \mapsto \{\texttt{"dsw\_il\_1"} \mapsto \texttt{position\_right}, \texttt{"dsw\_il\_2"} \mapsto \texttt{position\_left}\}.$$

Accordingly, applying the above to the function `RailML3_IL_SWITCH_BRANCHES` would yield the subset of `RailML3_IS_SWITCH_BRANCHES` that matches the position of the switch. In total, there are four possible combinations for the paths between A, B, C and D, in each of which a position must be specified for each associated interlocking switch.

4.3 Representation of Interlocking Objects

In order to validate safety properties of the animation model, it is important to have data on the interlocking system. This includes paths for blocks and routes, but also signalling. Since the interlocking schema was introduced with railML 3, there is not yet a publicly available example that implements a complete interlocking logic. It was therefore necessary to develop alternative solutions at some points in order to allow a meaningful validation of the model.

4.3.1 Blocks

Blocks in railways are usually used to maintain safety distances between operating trains and should thus only be occupied by at most one train. In railML 3, this concept is handled by train vacancy detection (TVD) sections, which are defined by *demarcating elements*.

These can be either buffer stops, open track ends or train detection elements, where open ends can be either implicitly demarcating or explicitly declared as “buffer stops”. Using the locations of these elements within the topology relation, the track parts of the TVD section can be determined. This is done in the computation of `RailML3_IL_TVD_SECTIONS` by looking at all pairwise combinations of its demarcating elements and adding the path across the topology in both directions between both elements. Here, the implicitly demarcating open ends are also taken into account, if there is a path to another demarcating object of the TVD section without an intermediate train detector. Compared to Abrial’s model, the TVD sections are also fixed like the blocks (i.e. no moving blocks). However, there is no restriction on the number of infrastructure elements per block as assumed in `ENV-4`.

Looking again at the example of Figure 4, consider a TVD section having the demarcating train detectors `tde1` and `tde2`. Then its parts are determined as the subset of `RailML3_IS_NET_RELATION_SUBSEQUENT_LOCATIONS` that are between `("ne1",0.65)` and `("ne3",0.1)` (both directions). If in addition `("ne2",0.0)` is an open end, the path from `tde1` to it is also included, since there is no other train detector between the two elements. The same applies, if the open end is explicitly marked as a demarcating element.

4.3.2 Routes

Routes are one of the most important assets of an interlocking system. They describe paths in the network that can be reserved by exactly one train at a time and then allow the train to travel safely along this path. Following the definition `ENV-6` of Abrial, a network always has a fixed number of routes. In contrast, routes here are characterised by exact paths between locations rather than abstract blocks. The TVD sections are only used as additional information about the vacancy of routes.

In railML 3, a route path is defined by its child elements `routeEntry` and `routeExit` (cf. Section 1.2). According to the documentation[14], they should refer to an interlocking object of a signal, such that routes are always delimited by signals, which also complies with Abrial’s definition `ENV-12`. However, there are additional special cases such as routes ending at buffer stops or starting at an open end. The route entries and exits are converted to the functions `RailML3_IL_ROUTE_ENTRY` and `RailML3_IL_ROUTE_EXIT`, which map all route IDs to the pair consisting of the ID of the referenced element and its spot location. This approach is also suggested by Martins et al. [Mar+22], who are dealing with formal verification of railML models against user-defined infrastructure rules.

The mere indication of the start and end of a route is usually not sufficient to define a unique path between them. Instead, as also stated in `ENV-7`, a route has to be additionally characterised by positions for intermediate switches that introduce the ambiguity. RailML 3 allows to specify these positions either for facing or for trailing switches. These are extracted into the variable `RailML3_IL_ROUTE_FORCED_SWITCH_POSITIONS`, which contains all forced

[14]`https://wiki3.railml.org/wiki/IL:route` (accessed on 25/9/2023)

switches positions (as introduced in Section 4.2) for each route. If the positions are correctly specified, which is ensured by the schematic validation, the unique route path can be derived and stored in `RailML3_IL_ROUTE_NXT`. First, the set of all possible paths between the entry and the exit is computed. Then, all branches of switch positions that do not match the forced positions are removed from this relation, so that only the correct path connects the entry and exit. Finally, the remaining parts of the other paths are cut off by restricting the relation to the parts that are reachable from the route entry.

The semantic validation ensures that the path continuously connects the inner locations between entry and exit by a total bijection (cf. `ENV-10`). Note that unlike for TVD sections, the resulting route paths are directed, and due to the assumption for the topology relation, the routes cannot contain cycles either. Moreover, some of Abrial's assumptions need to be weakened. In general, overlapping routes are allowed, and the first or last location of a route can be an inner location of another route (cf. `ENV-8`/`ENV-9`). The properties are still additionally checked, but only a warning is issued if one of them is violated.

These basic properties of routes, which largely align with Abrial's assumptions, are supplemented in railML 3 by a variety of additional data. One of these concepts is overlaps, which define one or more sections for routes that must also be kept clear to secure the sections that extend beyond the end of the route. The path of an overlap is computed in the same way as for routes in `RailML3_IL_OVERLAP_NXT`, except that the specified `relatedTrackAssets` and `isLimitedBy` elements are used instead. Each overlap can be linked to one or multiple routes to which it should apply. This is handled by the function `RailML3_IL_ROUTE_OVERLAPS`, which provides the set of overlap IDs for a given route ID. Required switch positions for the overlap can be specified using `requiresSwitchInPosition` together with a proof strategy (properties *must* or *should* be fulfilled, proved either *one-off* or *continuously* during overlap reservation), which is explained in more detail in the context of the animation in Section 5.2.1. Positions that *must* be proven are extracted to `RailML3_IL_OVERLAP_MUST_SWITCH_POSITIONS`. This also allows negated positions to be specified, i.e. a switch may be in any position except the specified one. Together with the proof strategy, this is modelled in B as a pair of type `RailML3_IL_REQUIRE_PROVING_TYPES * BOOL`, where the second value specifies whether the negated property must be proven (`TRUE`) or not (`FALSE`).

For both overlaps and routes, the required positions of all movable elements (derailers, movable crossings and switches) along the path are derived on B side and stored in the functions `RailML3_IL_{ROUTE,OVERLAP}_{DERAILERS,CROSSING_POSITIONS,SWITCH_POSITIONS}`. They are used in semantic validation for comparison of the specified and the derived positions, but also as guards for the route reservation process in the animation (Section 5.2.1).

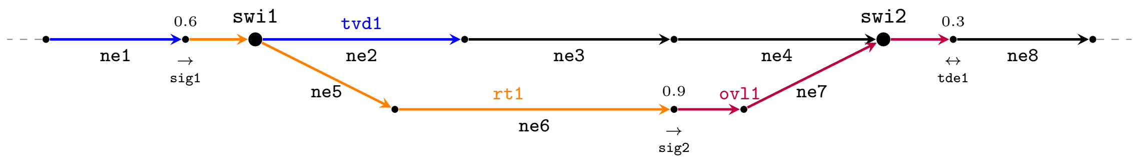


Figure 6: Concept of Route Paths and Overlaps

Example Figure 6 shows a simple example illustrating the route path of Listing 3 with its corresponding overlap and a TVD section. The `routeEntry` refers to `sig1`, the `routeExit` to `sig2`. In B, this is represented as (where `il_sig1` is the interlocking element of `sig1`):

$$\texttt{RailML3\_IL\_ROUTE\_ENTRY} = \{\texttt{"rt1"} \mapsto (\texttt{"il\_sig1"} \mapsto (\texttt{"ne1"}, 0.6, \texttt{direction\_normal})), \dots\}$$
$$\texttt{RailML3\_IL\_ROUTE\_EXIT} = \{\texttt{"rt1"} \mapsto (\texttt{"il\_sig2"} \mapsto (\texttt{"ne6"}, 0.9, \texttt{direction\_normal})), \dots\}.$$

Since the path between the two signals is already unique, the specified `facingSwitch-Position` would also be automatically derived from the path and is therefore not necessary. Specifying `position="left"` instead or any other not matching positions of further elements (derailers/crossings) would lead to a semantic validation error, as no route path between the signals can be derived considering this position.

The overlap `ovl1` is connected to the route `rt1` by specifying `hasOverlap` within the `routeExit`. On overlap site, this can be done by `activeForApproachRoute`, which will make the path of the overlap starting at the corresponding route exit. The end of this overlap is specified by `isLimitedBy="tde1"`. In order to set the overlap correctly, one could assume that the switch `swi2` is required to be in left position. This would be represented in B by the following element of `RailML3_IL_OVERLAP_MUST_SWITCH_POSITIONS`:

$$\texttt{"ovl1"} \mapsto \{(\texttt{"proving\_oneOff"}, \texttt{FALSE}) \mapsto \{\texttt{"swi2"} \mapsto \{\texttt{"il\_swi2"} \mapsto \texttt{position\_left}\}\}\}.$$

It expresses that the switch only has to fulfil the position once when reserving the overlap (one-off) and that the property is not negated (`FALSE`).

4.3.3 Signals

RailML 3 allows very detailed modelling of signalling by means of so-called signal plans consisting of `aspectRelations` for combining signal aspects of several signals. Compared to Abrial's model, the signalling is self-contained and basically independent of the routes. While this allows for a highly configurable specification, it also means that it is difficult to decide on the behaviour of a signal that is not explicitly specified. None of the example files available at development time (except the Simple Example with only two small example configurations of aspect relations) implements either a complete signal plan or an aspect relation. As a consequence, almost all signals in the formal model would be without any function. In order to avoid this issue, additional aspect relations are inferred from the specified routes, which in most cases are limited by signals (see previous Section 4.3.2). These inferred relations only use the three basic aspect types `closed`, `caution`, and `proceed` for simplicity, following Abrial [Abr10] (cf. `ENV-13`) and Gonschorek et al. [GBO18], as additional information would not currently affect the behaviour of the model. In ordinary aspect relations, however, all aspect types are supported. The only relevant information extracted from a signal aspect is whether a train is allowed to pass the signal or not. It is assumed that a signal is allowed to pass, if it does not have the `closed` aspect.

An aspect relation, as suggested by the railML schema, consists of three signal types: `slave` for the entry signal, `distant` for inner signals, and `master` for the exit signal, whose aspect usually influences the preceding slave and distant signals[15]. In B, the aspect relations are converted into records with one field for each of the three signal types, containing a function of the type

```
allIdsOfType("signalIL") +-> FIN(RailML3_IL_SIGNAL_GENERIC_ASPECTS),
```

which maps the signals to its set of aspects. In general, signals can have several aspects, for example the current aspect and an announcement aspect, so a set is needed here.

The specified aspect relations are stored in the function `RailML3_IL_ASPECT_RELATION-_SIGNAL_ASPECTS`, which maps the ID of an aspect relation to its relation encoded by a record as described above. The inferred relations are of the same type and are stored in the function `RailML3_IL_ASPECT_RELATION_SIGNAL_ASPECTS_INFERRED_FROM_ROUTE`, which maps a route ID to the corresponding set of inferred aspect relations. For a route with an entry and an exit signal and optionally additional signals enclosed by the route, all combinations of the two aspects `closed` and `passable` of the signals are considered, where all signals up to the first `closed` signal have the aspect `passable` and all subsequent signals are also closed. Here, the entry signal is always passable and assigned to the `slave`, the exit signal to the `master` and all inner signals to `distant`. Taking Figure 6 again as an example, these would be the inferred aspect relations of the highlighted route (if no aspect relation applies to the route):

$$\begin{aligned}\{\mathrm{rec}(&\texttt{master} : \{\texttt{"sig2"} \mapsto \{\texttt{aspect\_proceed}\}\},\\ &\texttt{slave} : \{\texttt{"sig1"} \mapsto \{\texttt{aspect\_proceed}\}\},\\ &\texttt{distant} : \emptyset),\\ \mathrm{rec}(&\texttt{master} : \{\texttt{"sig2"} \mapsto \{\texttt{aspect\_closed}\}\},\\ &\texttt{slave} : \{\texttt{"sig1"} \mapsto \{\texttt{aspect\_proceed}\}\},\\ &\texttt{distant} : \emptyset)\}\end{aligned}$$

For signals that are neither part of an aspect relation nor a route, an alternative strategy is required to give the signals a function. In general, a section must be defined for each signal that is controlled by that signal. This is done by searching for the closest signals in the direction of the signal, i.e. there is no other signal between the two signals.

[15] https://wiki3.railml.org/wiki/IL:implementsSignalplan (accessed on 27/9/2023)

The path between a signal and one of its closest signals forms the controlled section of the signal. For border cases, open ends and buffer stops are also considered as valid ends of control sections. This approach has similarities to the route detection algorithm proposed by Menéndez et al. [Men+23]. They also illustrate the concept in Figure 11, where the control section of signal *S22* is formed by the three closest following signals *T05*, *X15*, *S32*. For the implementation in B, this would result in the following pair for the function `RailML3_IL_SIGNAL_END_OF_CONTROL_SECTION`: `"S22"` $\mapsto$ {`"T05"`, `"X15"`, `"S32"`}. For each end, the path and its corresponding TVD sections are then computed in the same way as for routes (`RailML3_IL_SIGNAL_CONTROL_SECTIONS` resp. `RailML3_IL_SIGNAL-_CONTROL_TVD_SECTIONS`).

In general, only those signals are considered that have an associated interlocking object for control (`RailML3_IL_SIGNAL_CONTROLLED`). In rare cases this may lead to unexpected behaviour if a signal is ignored because its interlocking part is missing (cf. Section 10). It needs to be investigated whether this should be modelled differently during import or whether it is actually an error in the concrete model.

Finally, it should be noted that there are still some interlocking properties that are currently not taken into account. This mainly concerns the properties specified in the `signalBox` and `specificInfrastructureManager` sections, including, for example, various strategies for releasing the TVD sections. However, due to the lack of examples using these elements, their implementation has been deferred.

5 Animation

The animation of a B machine allows the user to check that the specification behaves as expected by interactively performing operations. In order to be able to study the behaviour of the railway system specified by a railML file, the `RailML3_animation.mch` machine contains several operations for animation of the model. As shown in Figure 7, this machine includes the import machine as described in Section 3.6 and uses only the derived relations as discussed in the previous Section 4. Although these are exported as variables by the rules machines, they are treated as constants, i.e. they are never changed after import, so that they can be converted to real constants when generating standalone machines (Section 6). The extending machine `RailML3_animation_init.mch` allows for specifying initial train positions within the topology and additional definitions for visualisation.

On the infrastructure side, the model allows the control of movable elements such as derailers, switches, and crossings. Interlocking features include routes, TVD sections, overlaps, signals, and signal plans for simultaneous switching of signals. In particular, the representation of the currently passable track and the route reservation process is again inspired by the train system model of Abrial [Abr10], but some simple signalling properties are also adopted. In this chapter, differences and similarities between the models are pointed out with reference to the assumptions and safety requirements formulated by Abrial.

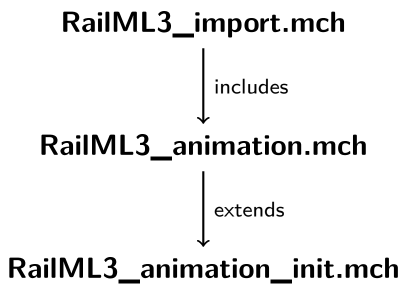


Figure 7: Machine Hierarchy of Animation

RailML is capable of modelling much more detailed properties in addition to the basic properties of Abrial's interlocking model. The model developed here for animating railML specifications is therefore an extension of this model. In general, the model has the same objective (`FUN-1`):

"*The goal of the train system is to safely control trains moving on a track network.*"

As for Abrial's model, some assumptions have to be made about the environment, the interlocking system, and the behaviour of the train. It should be clear that only safe train movements may take place in the system, i.e. the aim is that a safety-critical state such as a possible collision or derailment can never occur. Moreover, Abrial's assumptions about failures are adopted (`FLR-1` – `FLR-5`), i.e. trains cannot run over red signals or break down mechanically, and section occupancies are always detected correctly.

The following sections provide a more detailed overview of the possible operations, assumptions, and safety requirements that are encoded as invariants. One of the challenges is to write a model that works correctly for all railML specifications, as there are always different interpretations of the schema that the model should ideally cover.

This also complicates formal verification of the model in general, as deviations in a specification may lead to invariant violations that are not due to a modelling error. In this case, domain experts need to assess each violation manually. For any complicated formulas within invariants and operation guards, an additional description is provided by a pragma, which can be displayed by hovering over it in ProB2-UI. This should give domain experts a better understanding of what a formula actually expresses and why it may not be satisfied. A list of all operations with a brief description is provided in Table 10 in Appendix D. An overview of the interaction between operations can also be obtained from Figure 9.

5.1 Infrastructure

Firstly, all infrastructure components not implemented in the model are assumed to function correctly and automatically. This applies, for example, to the timely closure of level crossings (which can easily be added at a later stage), but also to the rarer case of tunnel gates. Also, `keyLocks` for locking of movable elements are generally disabled, as the movement of objects is currently only controlled by the interlocking system.

For each type of movable element, as described in Section 4.2, there are two basic operations, `IS_startChange<movableElementType>` to start the movement and `IS_endChange<movableElementType>` to complete the movement, where `<movableElementType>` refers to movable crossings, derailers, and switches. Splitting the start and end of the movement was made to allow for later refinement steps and to allow SimB to call the end operation after the typical throw time of the movable element (see Section 7 for a detailed description). The start event can only be executed if there is no train on the entire element, all TVD sections declared as belonging to it are vacant, and possible related movable elements are not in movement. For switches, the specified position restrictions for other derailers or switches must be satisfied for the new state. If the switch is a switch crossing, it is controlled as a single switch by the above operation. If two ordinary switches are coupled as related movable elements, their start should be triggered simultaneously, which is done by the additional operation `IS_startChangeCoupledSwitches`. The end operations can then be performed individually. Note that this only works properly if the switch types are specified correctly. As an additional constraint to reduce the state space, but also to simplify user interaction, a movable element can only change its state if:

- a train stands directly in front of it,
- it is part of a route/overlap in reservation (which is `SAF-3` of Abrial),
- it is part of a signal control section in front of whose associated signal the train is directly positioned or the corresponding signal is part of a signal plan in activation.

An element for which only the start operation was executed is in an undefined state. This is not modelled as an explicit state, but the destination position is kept in `IS_<movableElementType>sInMovement` during movement, so the previous and next position can be inferred, and also that the current position is undefined. The current states of movable elements are handled by the variables `IS_<movableElementType>_states`, which contain per movable element exactly one position of their precomputed positions (e.g. for switches in `RailML3_IL_SWITCH_BRANCHES`). If specified, the states are initialised with the preferred positions.

Having the positions of all movable elements, the passable and not passable branches can be inferred. To represent the current passable track, the not passable branches are removed from the topology relation `RailML3_IS_NET_RELATION_SUBSEQUENT_LOCATIONS` and stored in the variable `IS_next` (equivalent to `nxt` in the Abrial model). As each element can only have one passable part in total, the induced relation should be a partial injection, which is covered by a corresponding invariant. Additional invariants check that branches of elements in movement are not part of `IS_next`, just as for the positions that are currently inactive. Furthermore, it is required that the current position of all not moving elements is contained in `IS_next`, such as for switches:

$$\begin{aligned}&\forall sw.(sw \in \texttt{RailML3\_IS\_SWITCH\_IDS} \setminus \texttt{dom(IS\_switchesInMovement)}\\ &\qquad \Rightarrow \texttt{RailML3\_IL\_SWITCH\_BRANCHES(}sw\texttt{)(IS\_switch\_states(}sw\texttt{))} \subseteq \texttt{IS\_next}).\end{aligned}$$

Finally, the `keyLock` property is already prepared for all movable elements as a function that can later be controlled by the key lock elements. However, these are currently not imported from the railML file.

5.2 Interlocking

The interlocking system is crucial for ensuring safe train movements and thus constitutes the primary focus of the model. For this reason, at least the interlocking data for the TVD sections and routes must be specified in the railML file for correct animation.

As already mentioned in Section 4.3.1, Abrial's concept of blocks is transferred to TVD sections. A TVD section can be occupied or unoccupied by a train (`ENV-5`). The occupation of TVD sections is handled by the function `IL_occupiedTvdSections`, which stores for each occupied section the train that is currently occupying it. If a section is not part of the domain, this corresponds to a vacant section. When reserving blocks, the locations of the reserved sections are used as "blocks" rather than the TVD sections. This allows an accurate representation of reserved sections, even if no TVD sections have been specified for parts of a route. Such a block given by a location can be reserved by a route or an overlap, occupied by a train or free (cf. `FUN-3`). Reservation of blocks is managed by the set `IL_res_blocks` and the total relation `IL_res_route_blocks`, which contains all reserved blocks together with the routes that reserved them. Each block is allowed to be reserved for at most one route (according to `SAF-1`) with the exception of locations forming a route entry and exit. Those are allowed to be reserved by at most two routes, as otherwise, connected routes can never be reserved at the same time.

In contrast to Abrial's model, an occupied TVD section does not always have to correspond to reserved blocks (`FUN-4`), since in railML there can in principle be TVD sections that do not correspond to any route.

5.2.1 Routes

As mentioned, the modelling of routes is also inspired by the interlocking system of Abrial, which allows the adoption of some assumptions. The reservation process is controlled by several operations to represent the different phases of a route reservation. Since the railML schema does not provide a strict procedure for the route reservation process itself, it was necessary to define an approach for the behaviour of the model that takes the given data into account in a meaningful order. For this purpose, there are the two operations `IL_startRouteReservation` and `IL_endRouteReservation`. The former puts a route into the reservation phase, by adding the pair of its ID and the requesting train to `IL_routes_in_res`. If the route has overlaps, these are also added to `IL_res_overlaps`. The operation guard ensures, that this is only possible if there are no explicitly declared conflicting routes, other routes or overlaps intersecting the path of the route to be reserved, and if the TVD sections and blocks belonging to the route and associated overlaps are free and not reserved. Clearly, routes may be reserved by at most one train (`FUN-2`). In railML, a `routeActivationSection` can additionally be specified for each route, which means that a train can only request a route if it occupies at least one of its activation (TVD) sections. Due to the rare implementation of this property in the sample files, the activation sections are currently supplemented by the TVD sections of the route entry if none have been specified for the route. In order to prevent deadlocking when no activation sections can be derived at all, route reservation is currently also possible if

- the front of the train is located at the route entry,
- the route is an automatically locking route (`locksAutomatically="true"`),
- there is a signal plan in reservation for the entry signal, or
- the route entry corresponds to an exit of another route reserved by the same train.

By starting the reservation process, the blocks (locations) of the routes are reserved by adding them to `IL_res_blocks` and `IL_res_route_blocks` (cf. `FUN-5`). For movable elements that are part of the route or an associated overlap, it is now possible to change them until they are in the position that is required for establishing the route path (cf. `FUN-6`). If there are signal plans associated with the route, one of them must be activated and all associated signals must have the desired aspect change noted (see next Section 5.2.2 for details). During reservation, an invariant checks that none of the affected blocks is occupied by any train (cf. `SAF-4`). Once all movable elements are correctly set according to the route and overlaps, the changes of all signals of the associated activating signal plan are noted, and all additional route relations are fulfilled, the `IL_endRouteReservation` operation is enabled.

routeRelations can be used in railML for specifying constraints on positions or states of further elements such as switches for flank protection, but also for states of sections. The implemented features in B currently include "*must*" route relations on derailer and switch positions, as well as section states. These route relations *must* be fulfilled according to their proof method. They can be checked either once (`proving="oneOff"`) before the end of the route reservation, or `continuously` while the route is reserved. The one-off and continuous conditions are checked by the guard of `IL_endRouteReservation`, the continuous ones additionally by invariants. The same applies to the switch positions of overlaps (Section 4.3.2). While it is quite easy to formalise the route relations that *must* be proven, railML also allows properties that *should* be fulfilled. Including them in the corresponding guard and invariants would turn these conditions into *must* conditions, which is too strict. An alternative approach could be to probabilistically check the fulfilment of the property by simulation while the route is reserved. More information about proof strategies and the state space can be obtained from the railML wiki[16].

Once the route reservation is completed, all movable elements involved are locked by the route to ensure that the route path remains set. This is done by adding the element IDs together with the route ID to `IL_{crossing,derailer,switch}_locked_routes`. Then, the route is moved from `IL_routes_in_res` to `IL_res_routes`, which corresponds to a formed route in Abrial's model and allows the reserving train to travel along the route (cf. `FUN-7`). Invariants are used to check that locked elements are never in motion and that for all reserved routes the current positions match the required positions of the route. In addition, routes may not be in reservation and reserved at the same time. The model also monitors several safety requirements of the reserved routes through invariants:

- all reserved routes are disjoint,
- no other train except the reserving one is on the route,
- for each reserved overlap exists a corresponding reserved route,
- each reserved block corresponds to a reserved route,
- all route blocks of a reserved route (except those released by partial route releases) are part of the reserved route blocks, and
- the successors of all reserved blocks of a route are also reserved by the same route.

For the release of routes, there are different ways in railML 3. Routes can have `routeReleaseGroups`, consisting of TVD sections, in front of the train (`ahead`) and behind the train (`rear`). A `routeReleaseGroupRear` can be released using the operation `IL_partialRouteReleaseRear` if the train has passed the associated TVD sections completely. The release is performed sequentially, starting with the rearmost release group, until the last group is released after the train has completely cleared the route. The already released groups are kept track of per route in `IL_released_partialRoutes`. Once all rearward release

[16] https://wiki3.railml.org/wiki/Dev:StateSpace (accessed on 15/10/2023)

groups of a reserved route are released, the complete route can be released by the operation `IL_completeRouteReleaseRear`. If a route does not have any release groups, the complete release is directly possible as soon as the train has passed the complete route.

The rearward release of the routes roughly complies to the assumptions of Abrial in the way that the complete route remains reserved as long as some release groups are not released or the train occupies parts of the routes (`MVT-1/2/3`). Deviating from this (and `TRN-4`), it is also possible to release routes where the train is still running. This can be useful if a train has stopped and no longer needs the rest of the previously reserved route after a change of direction. For this purpose there are `routeReleaseGroupAheads` in railML 3, which can be released by `IL_partialRouteReleaseAhead` if the corresponding TVD sections have not yet been occupied by the reserving train. The complete route can be released if all release groups ahead have been released and the train is currently on a TVD section that is a berthing track (to change direction). In all other cases, it is assumed that the train will continue its journey, so the route will eventually be used.

With each release operation, the movable elements that are part of the released sections are unlocked and all affected blocks are no longer marked as reserved. The routes themselves stay reserved until a complete release is performed. Together with the complete release of routes, all associated reserved overlaps are also released. While a route is reserved, it is also possible to trigger the release of overlaps separately with the operation `IL_startOverlapRelease` if the train is on one of the `releaseTriggerSections` that can be specified for the `overlapRelease` elements. The overlap is marked for release by adding its ID to `IL_overlaps_in_release`. The release process can be finished by executing `IL_endOverlapRelease`. Splitting of these operations allows the `overlapReleaseTimer` to be used in the SimB simulation.

5.2.2 Signalling

As already mentioned in Section 4.3.3, signalling examples in railML 3 are rarely available. Therefore, as for the modelling, an alternative solution for the signal behaviour had to be developed to make the model animatable with meaningful properties, as signals are the only elements that can prevent a train from entering a reserved or occupied section. The approach will need to be reviewed as more, and especially complete, examples of signalling become available.

In general, two types of signals are considered. On the one hand, those that are part of a signal plan (used here synonymously for aspect relation) and on the other hand, those that are controlled individually without a signal plan (`RailML3_IL_SIGNAL_NOT_CONTROLLED_BY-_SIGNALPLAN`). The state of all signals is kept in `IL_signal_states`, which maps the IDs of controllable signals to the set of their currently showed aspects. It is assumed that each signal will fall back to `{aspect_closed}` as soon as the front of a train passes it (`ENV-15`).

To change the signals controlled by a signal plan, such a plan must be in activation. This can happen with or without an associated route in reservation by execution of the operation `IL_startActivateSignalplan`. The operation is enabled if the desired signal plan is not in activation, there is no conflicting signal plan using the same slave or master signal, and if the slave signal is the master signal of a plan that is currently in activation, the aspect must be the same as for the other signal plan. In the current models, signal plans are always activated after a corresponding route reservation has been started (i.e. after execution of `IL_startRouteReservation`), since they are always associated with routes. The signal plan to be activated is then copied to `IL_signalplans_in_activation`, where all signal plans currently in activation are stored as a set.

A signal plan already defines the target aspects of the signals after its activation. To keep the change of signal aspects during activation without applying the changes directly, there is the concept of `IL_noted_signal_states`. Otherwise, signals may switch to a passable aspect too early, especially in the case of unfinished route reservations, if the route path is not set correctly. A change for a signal that is part of a signal plan in activation can be "noted" using the operation `IL_noteChangeSignalState`. This is possible if the track currently set in `IS_next` is not occupied by a train up to at least one end of the signal's control sections. If the master signal is intended to show a passable aspect, it must be ensured that at least one of its sections is free and passable. Therefore, its aspect can only be noted if either a subsequent signal plan with a slave with the same aspect has been successfully activated, or if the master signal is an individually controllable signal and all conditions for entering its section are fulfilled. Otherwise, the master signal can never be changed and the activation of the signal plan gets stuck. After change of a movable element within one of the control sections of a signal, the noted signal states for that signal are reset to check that the conditions are still met after the change. If the signal plan was activated together with an associated route, the route reservation first has to be completed before the activation of the signal plan can be completed. This ensures by the conditions of the route that the set track is passable before the noted signal changes are applied. Since all slave signals of the signal plans inferred from routes have a passable aspect, the final activation of the signal plan by executing `IL_endActivateSignalplan` corresponds directly to `FUN-8` and `SAF-2` of Abrial. All signals that have a passable aspect except the master signal, which may be controlled by another subsequent signal plan, are added to `IL_signal_locked` to prevent them from being changed individually. The lock is removed once the train has passed the locked signal.

For the individually changeable signals, the signal sections derived in Section 4.3.3 are used. For these signals, only the aspects `{aspect_closed}` and `{aspect_proceed}` are considered for simplification. Initially, all signals show the closed aspect. If for at least one section of the signal all associated TVD sections and locations are vacant, and the entire path to the end signal of that section is correctly set by the movable elements, a signal can be changed to the passable aspect. This is roughly equivalent to a "loose" route, but one that can be released at any time by occupation of another train, starting conflicting route reservations, or changes of a movable element. In one of these cases, all associated signals of the affected sections immediately fall back to the closed aspect. When a train occupies parts of a signal section, the movable elements cannot be changed until the train has passed them, to ensure that the track remains passable. To avoid state space explosion, independent signals can only change when a train is directly in front of it or it is the master signal of a signal plan in activation. As a special case, not locked distant signals of an activated signal plan can also be changed individually to avoid trains getting stuck in a signal plan with closed aspects for some of the distant signals. In this way, the train can proceed to the master signal (provided all conditions for passability are met), which is then controlled again as usual.

Like the other interlocking objects, the signals must fulfil some requirements for safe operation, which are covered by invariants. These are almost the same for the actual and for the noted signal states:

- if `aspect_closed` $\in$ `IL_signal_states(sig_id)`: it is the only aspect of the signal (otherwise its aspect would be ambiguous)
- if signal is a slave or distant signal of a signal plan in activation with passable aspect: ID of corresponding infrastructure signal is locked (i.e. in `IL_signal_locked`)
- if `aspect_closed` $\notin$ `IL_(noted_)signal_states(sig_id)`: none of the locations of the currently set track (`IS_next`) is occupied; only for `IL_signal_states`:
  - movable elements after the signal form a passable track to the exit signal
  - TVD sections of the currently set track are not occupied

The last two properties are only required for the actual signal states, as switches are still able to move during a route reservation, which can lead to an incorrectly set track. At the end of the reservation, it is ensured that the route is passable, so that this property is also guaranteed here.

5.3 Train Movements

The model also allows train movements similar to Abrial's model, with separate events for moving a train forwards and backwards (`RS_trainMoveFront`, `RS_trainMoveBack`). By the guards of both events it is always possible for a train to move, except the one and only case that the front of the train has the same location as a signal with a closed aspect, i.e. `aspect_closed` $\in$ `IL_signal_states(sig_id)`. Following `ENV-13`, trains are assumed to stop correctly at closed signals and are equipped with a train protection system that prevents them from running over closed signals. A single train movement step is represented by changing the current position from the previous location to the next location of `IS_next`.

For each train, its front and back position are stored in RS_trainFront/-Back. All occupied locations are stored per train in RS_trainOccupiedLocations without direction, as this is irrelevant for the occupation status. With the help of this, invariants are used to check that there are no colliding trains (i.e. the intersection of the occupied locations is not empty for at least two trains) and that trains have not broken into two parts, for example by crossing an incorrectly set switch, which could lead to a derailment:

$$\begin{aligned}&\forall t.(t \in \texttt{RS\_arrivedTrains} \cup \texttt{dom(RS\_requestingArrivalTrains)}\\&\quad\Rightarrow \texttt{RS\_trainBack(}t\texttt{)} \mapsto \texttt{RS\_trainFront(}t\texttt{)} \in \texttt{closure1(IS\_next)}).\end{aligned}$$

The assumption TRN-1 applies as well, meaning that trains are considered as indivisible units that cannot split. While TRN-2 states that trains are not allowed to move backward in general, trains in railML are allowed to change their direction on designated berthingTracks. If a train is completely on one TVD section that is declared to be a berthing track and has released all its reserved routes (to avoid them being blocked forever), it can change its direction using the event RS_trainChangeDirection.

As the train moves, it is checked if it has passed a train detection element (e.g. axle counter), which always works correctly according to the assumptions about failures. If the front of the train passes one or multiple train detectors in a movement step (i.e. the new position is the next position behind the detector), the associated TVD sections are marked as occupied by adding its ID paired with the train to IL_occupiedTvdSections. In the opposite case, a TVD section is declared vacant again as soon as the end of the train has moved to an associated detector and this is also the last one in the section. In some cases of incorrectly configured train detectors or TVD sections, this can lead to a failure to detect that a section has been freed. Then there is an occupied TVD section with no train on it, which should not happen and is therefore covered by an invariant.

Currently, the number of trains is influenced by the set RS_trains. For the future, once sample files are available, it is planned to import the possible set of trains, perhaps together with other additional properties, from the rollingstock schema. A distinction is made between trains that have arrived (RS_arrivedTrains), trains that are requesting for arrival (dom(RS_requestingArrivalTrains)), and trains that have not arrived (RS_trains without the other trains). All these types of trains are disjoint.

A train can announce its arrival at one of the open ends of the track and request to occupy the track using the operation RS_trainArrivalRequest. This requires that the open end demarcates a vacant TVD section. Together with the location of the open end, the train is added to RS_requestingArrivalTrains and the corresponding TVD section is marked as occupied by the train. On the interlocking side, a decision can now be made to accept (IL_trainAcceptArrival) or decline (IL_trainDeclineArrival) the arriving train. If there is a route that starts at the open end, the reservation of this route must have been started before the acceptance. If the train is accepted, it becomes a arrived train, its back is set to the open end and its front to the next location after the open end. If its arrival is declined, the TVD section is simply made unoccupied and the train is removed from RS_requestingArrivalTrains.

The counterpart to the arrival operations is the operation `RS_trainLeave` for leaving trains. This comes into play when a train has fully approached an open end. Performing this operation removes the train from the `RS_arrivedTrains`, immediately releases all its reserved routes, and frees its remaining occupied locations. After that, the train is no longer controlled by the system.

As one does not only want to examine scenarios with newly arriving trains, but perhaps also trains that are already in the system at certain locations, it is possible to influence the initialisation of the train positions by using the constants `RS_trainFront_init` and `RS_trainBack_init`. Here only the exact positions for the front and back of the trains need to be specified. Further details such as the occupied TVD sections are automatically computed in the initialisation.

6 Generation of Standalone B Machines

Although the animation machine based on the import process can be used directly by specifying the file path, it may be interesting to generate machines that persist the data of one model and can be run independently. The main motivation is to improve the performance, as the operations of the import process no longer need to be computed for each state, and the variables generated by the rules machines can be converted to constants. In addition, it may be useful to be able to save the model so that it can be examined at a later stage without having to import it again. Due to the current limitations of the B-Rules DSL parser, a rules machine cannot include the import machine, which makes it impossible to formulate custom rules on top of the imported data. Fortunately, this can be circumvented by generation of a separate rules machine for validation, which is described in Section 6.2. Finally, the procedure explained in this chapter is applied in the ProB2-UI plugin described in Section 9 for the creation of the machines.

6.1 The Generation Process

The generation process is also handled by machines written in classical B, and uses the external function `FPRINTF` of ProB to print the imported data to designated files. This has the advantage that data structure values can be processed directly in B itself and output with the appropriate type information without going through the ProB Java API. The general concept is depicted in Figure 8. As for the animation, there is one machine for the actual logic (`RailML3_printMachines.mch`), and one that extends this machine, setting necessary properties for constants like the `file` as usual, but also individual machine names and corresponding paths of the output machine files.

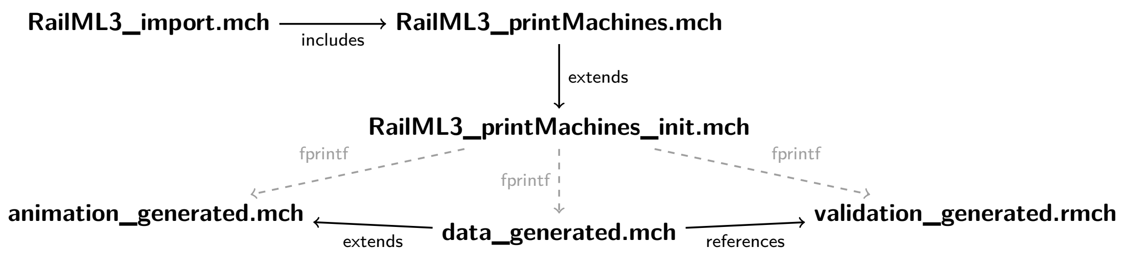


Figure 8: Machine Hierarchy of the Generation Process

The `RailML3_printMachines.mch` machine contains three operations for printing of *data*, *animation* and *validation* machines according to the railML input file. The data machine contains all the data structures created during the import process. The animation machine contains the same code as the machine described in Section 5, except that the inclusion of the import machine and its operation `importRailML` are omitted, and the data machine is extended instead. In the generated validation machine, custom rules can be specified based on the data present in the referenced data machine.

Before printing, the machine files should be empty, as FPRINTF currently only appends the output to the contents of a file, and otherwise undesirable results may occur. Before printing the data machine, it is also checked that there are no invariant violations or other import related errors by checking that "FALSE" $\notin$ ran(FORMULA_VALUES("inv")) and no_error = TRUE. The animation and validation machines cannot be printed until the data machine has been printed, as both require it. To prevent the machines from being printed without the corresponding operation being triggered, there are three trigger operations that set the guard of the operation to print the selected machine type to TRUE. As the operation is evaluated before execution, printing will take place immediately after a trigger operation is executed. This does not apply if the machine is loaded using the ProB Java API. In this case, the corresponding print operations must also be executed. With this exception, the RailML3_printMachines.mch machine is designed in such a way that it can be used without modification as a resource for the ProB2-UI plugin (Section 9), but also in the setup shown here with an extending machine.

Data Machine In the data machine all records of imported railML types and all derived data structures are stored (Section 3). In contrast to the rule-based machine, the data in the generated machines can be accessed and changed directly via the data machine file. This can be beneficial if the user wants to try out minor modifications without a complete re-import. All imported records are stored so that the user can access as much data as possible when creating custom rules in the validation machine. For the animation machine, a small subset of the derived data structures is sufficient. It extends the data machine for simplicity, as only one data machine needs to be generated this way. For performance reasons when setting up constants, it may be interesting to use a second reduced data machine containing only the data needed for the animation.

A crucial point in the generation of the data machine is that the variables created by the rules machines are converted into constants. To do this, the variable identifiers are obtained from FORMULA_INFOS("variables") and printed comma-separated in the CONSTANTS section of the generated machine. The PROPERTIES section of the machine is mainly generated by the specially created external function VARS_AS_TYPED_STRING, which outputs all matching variables together with their current content including type information as one parsable string for a given prefix (here: “RailML”). To prevent the data from being printed in one long line, additional line breaks are introduced at the & symbols. However, there can still be long lines, which might cause problems with some editors. Unfortunately, for very long sets, ProB currently also reaches an internal limit, which causes the set to be truncated at some point and replaced by “...” in the output. Both issues need to be investigated further on ProB side, although fortunately the latter only affects a few models.

Table 1: Machine Statistics for Rule-Based and Generated Animation Machine

	Rule-Based	**Generated**
Files	17	10
Deferred Sets	1	1
Enumerated Sets	49	48
Definitions	243	200
Constants	8	201
Variables	580	30
Properties	13	204
Invariants	618	70
Operations	257	26

A special case of a created variable is the precomputed transitive closure of the topology relation `railML3_IS_NET_RELATION_SUBSEQUENT_BLOCKS`. Obviously it is not useful to store the transitive closure in the generated machine. Therefore, the name of the variable starts with a lower case letter so that it is not captured by the prefix that outputs all generated railML sets, and the equivalent constant is computed by the generated machine:

```
railML3_IS_NET_RELATION_SUBSEQUENT_BLOCKS_closure1 =
    closure1(railML3_IS_NET_RELATION_SUBSEQUENT_BLOCKS).
```

In addition, the two constants `allIdsOfType` and `all_ids`, which contain all IDs, are copied. At the end, the required enumerated sets for the railML attribute types are printed.

Animation Machine To print the animation machine, the contents of the `RailML3_animation.mch` machine are copied (without using `importRailML` in the initialisation) into the `printAnimationMachine` operation. This has the disadvantage that any change made to the machine must also be transferred to the print machine. The resulting animation machine has almost the same content as the rule-based variant, and extends the data machine rather than importing the data. Depending on the variable `LINK_SVG`, it can be controlled whether the VisB definition file and the corresponding SVG file should be added to the animation machine. By converting the variables to constants and omitting the import operations of the rules machines, the generated animation machine has significantly fewer invariants, as can be seen from the comparison of the machine statistics in Table 1. This can result in benefits for model checking but also for the general performance of the animation. Also, as mentioned above, not all of the loaded constants are used by the animation machine, where further improvement could be made. With the exception of the set for supported railML versions, the enumerated and deferred sets are the same as for rule-based animation. The number of files and definitions is mainly due to the additionally loaded definition files.

6.2 Validation of Custom Rules

Users may wish to formulate additional rules that are not covered by the semantic validation rules during import. The B-Rules DSL can be used for this purpose as well. As already mentioned, this is currently not possible on top of the import machine, as rules machines cannot reference ordinary B machines that contain variables or operations. Having the generated data machine instead, it is possible to reference it from within any rules machine, as the data machine only specifies constants and their properties. In principle, this is already sufficient to validate custom rules. The generated validation machine only comes with a few useful additional definitions, such as `distanceBetween` for the distance between two locations with respect to the topology. Currently, this only works for unique paths between two locations. It is planned to extend the definition for arbitrary locations by using more advanced definitions or abstract constants (e.g. to implement a breadth-first search). However, recent attempts with different approaches failed due to an unacceptable increase in runtime.

In addition to these definitions, some example rules are provided that focus on properties that are not subject to semantic validation, but may be of interest when dealing with regulatory requirements or standards that the specification must meet. Luteberget et al. [LJS16] and Martins et al. [Mar+22] provide inspiration for technical rules such as that each entry signal of a route should be at least 200 metres before the first facing switch, that each signal should have an approach speed less than a maximum value, or that train detectors should be at least a minimum distance apart. The latter is checked by the example rule `MinDistanceOfTDEs`, for which such a distance can be configured individually. The second example rule, `EBO_CheckSwitchSpeeds`, checks that the specified maximum allowed speed for switch branches does not exceed the value prescribed by the German “Eisenbahn-Bau- und Betriebsordnung” (EBO)[17]. Similar rules can be written for regulations concerning the speed, the maximum radius or inclination of a track curve (`speedSection`, `gradient/horizontalCurves`).

As a current limitation it must be noted, that the custom rules can only be applied to railML elements that are actually imported. It is of course possible to export the constant `data`, which contains the entire railML file. However, formulating rules using `data` requires knowledge of the internal processing of XML files in ProB and is therefore not as user-friendly. The corresponding line that performs the export can be found commented out in the `RailML3_printMachines` machine if required.

For convenient checking of the rules, it is recommended to use the “Rule Validation Language Plugin” available for the ProB2-UI[18]. Further details on the plugin are provided by Heinzen [Hei18].

[17] `https://www.gesetze-im-internet.de/ebo/__40.html` (accessed on 20/10/2023)
[18] `https://github.com/hhu-stups/prob2_ui_rule_validation_plugin`

7 Simulation

The properties of the generated models, especially the invariants, should ideally be automatically verifiable. Classically, this is done by model checking, where the entire state space is explored and checked for erroneous states or invariant violations. However, this technique suffers from the problem of state space explosion. Unfortunately, it turns out that this is also true for most of the railML models, so this approach is not applicable.

In order to be able to perform an automatic investigation of the properties, SimB [VLM21] offers the possibility to perform timed probabilistic simulations with B-models in ProB. Statistical tests can then be used to validate properties of the model by running several Monte Carlo simulations with different parameters as also proposed by Cappart et al. [Cap+17]. This approach is a compromise between full model checking, which is not possible due to the size of the state space, and single simulations, which do not provide probabilistic statements about the properties to be checked.

More specifically, SimB allows the investigated time interval of Monte Carlo simulations to be specified by start and end predicates, a fixed number of steps, or a duration. Hypothesis tests allow statistical statements about the fulfilment of invariants or individual predicates, but also about the maximum elapsed time between two statements (timing). Estimators can be used to make further statements about averages or cumulative sums. In addition, real-time simulation is possible to study an approximate realistic behaviour of the model.

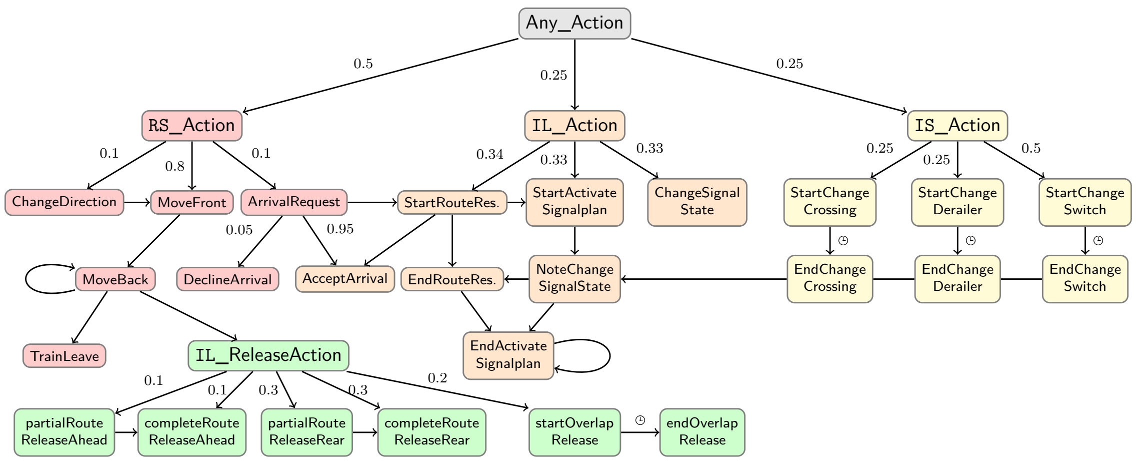


Figure 9: Concept of the SimB Activations

The first step is to configure the simulation, i.e. the order in which operations are activated, in a JSON file. This should be based on the general intended behaviour of the model and, where appropriate, assumptions of probabilities. Here, for the animation machine, the activations are configured as outlined in Figure 9. From the generic root activation `Any_Action`, the next activation is chosen by probabilistic branching into rolling stock, interlocking, and infrastructure actions. Again, probabilities determine the actual operation

Listing 18: SimB Activation Configuration for End of Movement of a Derailer

```
{
    "id": "IS_endChangeDerailer",
    "execute": "IS_endChangeDerailer",
    "after": "RailML3_IL_DERAILER_TYPICAL_THROW_TIMES(DerailerId)",
    "fixedVariables": { "DerailerId": "DerailerId" },
    "activating": ["skip", "IL_noteChangeSignalState",
       "IL_endRouteReservation"]
}
```

to be executed. For example, a train movement should be performed more often (80%) than a change of direction (10%). For the selected operation, it is checked if it is enabled by another probabilistic activation and if not, `skip` is activated. If it is enabled, the operation is activated and also corresponding subsequent activations, such as the end of a route reservation following its start operation. The top activations therefore always trigger certain procedures, such as route reservation or train movements. Each executed activation also activates the `skip` activation, which in turn activates `Any_Action`. In this way, the selection process of the next operation is permanently executed during the simulation.

The parameters of activations without having any fixed variables are chosen uniformly by `"probabilisticVariables": "uniform"`. The default value for the continuation of time (`after`) is 1000 ms. For certain activations, the time span defined in railML is used (🕒 in Figure 9). These include the movable elements, for which their `typicalThrowTime` attribute is used as the time between activation of the start event and the end event, but also the `overlapReleaseTimer` for the time between the start and end of an overlap release. Listing 18 shows the configuration of `after` for the end of a derailer movement after its typical throw time. To ensure that the operation is executed after the delay for the same derailer that caused the activation by starting the movement, the variable `DerailerId` is fixed to `DerailerId` (the latter being the ID of the executed operation that triggered this activation).

For the train movement, it is planned to model the train speed in terms of the permitted speed of the current speed section and the length of the current net element in combination with the intrinsic position of the train. However, this approach could not yet realised in SimB as it is not possible to pass the required information to the `after` of the initial phase of movement (`RS_trainMoveFront`). For the rear part of the train this would be possible, but could lead to situations where the front part is moving much faster than the rear part of the train. In order to slow down the movement of trains compared to typical throw times of movable elements, the default delay is set to 5000 ms.

With this configuration, different types of simulation can be carried out. On the one hand, a real-time simulation can be used to check whether the behaviour of the model is as expected. The current configuration also provides limited support for interactive simulation [VL23], for example, if the user starts the movement of a movable element, the movement is completed automatically after its typical throw time. On the other hand, Monte Carlo simulations with statistical checks can be performed on multiple traces to make statistical statements about temporal or probabilistic properties. A common check is a Monte Carlo simulation run for a certain number of steps or time, using a left-tailed hypothesis test to examine the probability of an invariant violation:

```
SIM(ending: e.g. after 300 steps or 20000 ms
    property:  ALL_INVARIANTS
    check:  HYPOTHESIS
    procedure:  LEFT_TAILED
    probability:  0.999
    α:  0.001).
```

Although this approach cannot replace complete model checking, it can help with validation and uncover any issues with the model, should the hypothesis be rejected (i.e. the probability that all invariants are true is lower than assumed). It should also be noted that the simulations are already subject to certain assumptions due to the configuration of the activations, which may result in certain scenarios not being covered.

Besides the invariant check, specific properties can be examined, including that a route becomes eventually reserved or that a train has passed a certain position after a certain time. Further research could attempt to incorporate data from the timetable schema into SimB, and based on this, check whether a train position matches the expectation of the timetable. Another interesting property that can be checked is whether the movements of a movable element always end within the maximum time allowed for the element. To do this, the start and end predicate to delimit the movement period and the timing property must be set to the `maxThrowTime` of the element:

```
SIM(starting: "swi1" ∈ dom(IS_switchesInMovement)
    ending: "swi1" ∉ dom(IS_switchesInMovement)
    timing: RailML3_IL_SWITCH_MAX_THROW_TIMES("swi1")
    check:  HYPOTHESIS
    procedure:  LEFT_TAILED
    probability:  0.9
    α:  0.1).
```

Unfortunately, one problem with the simulations is that they become quite slow for complex topologies. This can be due to expensive operation guards that need to be computed in each state, but also to the probabilistically uniform selection of the next transitions. Therefore, a small value for the ProB preference `MAX_OPERATIONS` is recommended, as then fewer transitions need to be computed in each state (to avoid problems with switch crossings and their positions, the value should be at least 2).

8 Visualisation

Visualisations can help to identify errors in a formal model that might not be noticed simply by looking at the data. VisB [WL20], a visualisation technique based on Scalable Vector Graphics (SVG), is available in ProB for this purpose. It allows an individual visualisation of the current state of a machine and, in ProB2-UI, the execution of events triggered by user interaction with a specific element within the SVG. To create such an SVG, the implementation of *Graphviz* [Gra23] included in ProB is used. Graphviz is a tool for visualising graphs and therefore well suited to a topology-based model such as railML 3. There have also been attempts by railML.org to visualise pure relational topology data in graph form with Graphviz [Kol23]. In contrast, the aim here is to create a view that is more similar to an interlocking view, in order to better represent train positions and further properties in VisB.

Since SVGs created with Graphviz cannot be used directly for VisB, certain conversion steps are required. An overview of the necessary steps is provided in the following sections.

8.1 First Step: Create SVG Using Graphviz

As mentioned, ProB supports creation of so called *custom graphs*, which can visualise the current state of a machine in form of a customised graph. The custom settings are affected by the special definition `CUSTOM_GRAPH`, which is part of ProB since version 1.12.2 and combines the former definitions `CUSTOM_GRAPH_NODES` and `CUSTOM_GRAPH_EDGES`. Since this version, the attributes of nodes, edges, and the graph itself can also be easily controlled via records. A simple exemplary definition can be found in Listing 19.

Listing 19: Example of a `CUSTOM_GRAPH` Definition

```
CUSTOM_GRAPH == rec(
    layout: "dot",
    directed: FALSE,
    nodes1: UNION(i).(i : 1..2 | {rec(shape: "circle", nodes: i)}),
    edges1: {rec(edge: 1 |-> 2, label: "1_2")})
```

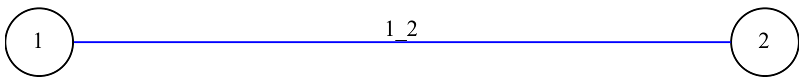


Figure 10: Custom Graph Created by the Definition in Listing 19

It describes a graph with two circular nodes and an edge connecting them, as shown in Figure 10. Within the `CUSTOM_GRAPH` definition it is possible to specify the graph attributes, as is done here for the `layout` engine. The entry `directed` is a virtual attribute to instruct ProB whether the generated graph should be directed. In the case of `TRUE`, a `digraph` would be created instead of a `graph` for undirected graphs otherwise.

In the context of visualising railway lines, undirected graphs are usually more appropriate, which is why they are used in all the following graph definitions.

For the visualisation of railML 3 topologies, three predefined definitions of custom graphs are available in `RailML3_CustomGraphs.def`. The definition `DOT_customGraph` attempts to visualise the topology using only its relational data and the placement algorithm of a Graphviz engine, without using any positional data. The others are for two common ways of storing position data in the railML `visualization` schema, namely linear position data as used in the railML files exported by NEAT's D4R Track Planner[19] software (`D4R_customGraph`) and position data based on intrinsic coordinates of net elements as used in the Norwegian Railway Directorate's railML files exported by railOscope[20] (`NOR_customGraph`). The strategy to be used depends on the file, which means that applying a strategy to a file that was not created using that strategy is unlikely to result in a correct visualisation.

To create the visualisation, it is sufficient to load the definition file in the `RailML3_animation_init` machine (Section 5) and set the `CUSTOM_GRAPH` definition to one of the available definitions. The three definitions are further elaborated in the following sections.

8.1.1 Graph Derived from Infrastructure Data (`DOT_customGraph`)

In order to derive a graph from the topology representing the connection between the infrastructure elements of the track ends, an additional relation is required. This is implemented by `RailML3_IS_TRACK`, which stores the relationship between net elements and intermediate or delimiting infrastructure elements of a net element such as switches, crossings or buffer stops. If the track ends are not explicitly specified, they are inferred from the track paths and the elements at the beginning and end of the track. An exemplary track `trc1`, which starts in a switch `sw1`, then leads via a net element `ne1` to a crossing `cr1` and finally via `ne2` to a buffer stop `bus1`, would be represented as

$$\texttt{"trc1"} \mapsto \{\texttt{"sw1"} \mapsto \texttt{"ne1"}, \texttt{"ne1"} \mapsto \texttt{"cr1"}, \texttt{"cr1"} \mapsto \texttt{"ne2"}, \texttt{"ne2"} \mapsto \texttt{"bus1"}\}.$$

This is already all the data needed to create the graph. Note that in the topology given by railML, the net elements represent the nodes and the net relations represent the edges. When visualising railway lines, it is usually desirable to represent the net elements as edges instead, so that train positions can be represented on them. To do this, the custom graph definition cannot use the given data directly, but must interpret the nodes as edges and the relations as nodes. This is done by creating nodes whose identifiers consist of the IDs of the two net elements that are part of a relation pair in `RailML3_IS_NET_RELATION`. Their uniqueness is ensured by ProB's external function `SORT`, which sorts both elements of the pair according to their internal order. As shown in Listing 20, the resulting sequence of strings is concatenated as the node ID. The edges connect these relation points to each other

[19] https://design4rail.com/service/d4rhorizon/#section-trackplanner (accessed on 29/10/2023)
[20] https://railoscope.com (accessed on 29/10/2023)

Listing 20: Creation of Virtual Nodes for NetElements in `DOT_customGraph`

```
rec(shape:"point", nodes: STRING_CONC(SORT({i_ne,i_ne2})),
   ‘id‘: STRING_CONC(SORT({i_ne,i_ne2})), width: "0.0")
```

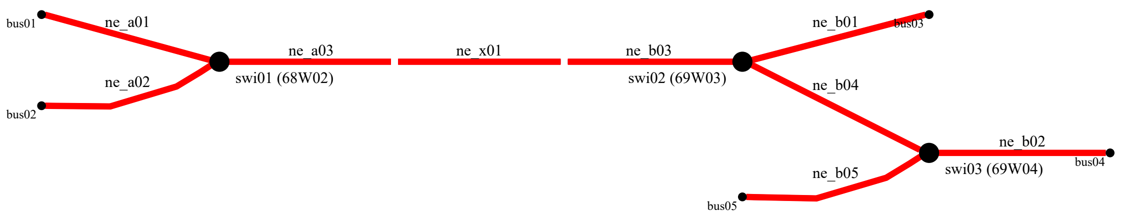


Figure 11: Simple Example Visualised by Graphviz Without Positioning Data

if the relation pair is not part of a switch or a crossing, and otherwise to the ID of the switch or crossing. If the track ends at a buffer stop, an edge is also added to it. When creating the edges, it is important to note that they are created in the correct direction from the intrinsic coordinate 0.0 to 1.0, so that the later visualisation with linear gradients, which require the percentage information for the display of positions, works properly. This is ensured by the orientation of the tracks stored in `RailML3_IS_LINEAR_LOCATION_KEEPS_ORIENTATION`. In the case of the orientation `FALSE`, the edge is created inversely.

This technique does not currently support the visualisation of signals and other functional infrastructure on net elements, as this requires the coordinates of the edges to be known in order to place the nodes accordingly. Since the placement of the edges is done internally by Graphviz, the required data is obviously not accessible. One could consider adding such points as nodes, which would require a more sophisticated graph definition and might make the graphs too complex. There have also been attempts to use the linear coordinates given by linear positioning systems to infer at least one coordinate of the elements, but fixing of only one coordinate is difficult to handle in Graphviz. To get around this, attempts were made to precompute offsets for overlapping elements in B, but this took a lot of computational time and resulted in unsatisfactory visualisations.

An example of a railML file which comes without visualisation data for the topology is the Simple Example. Using the definition `DOT_customGraph` with `layout:"dot"` gives the visualisation shown in Figure 11, which represents the actual topology very well. Another example can be found in Figure 16b in Section 10.

As seen, this visualisation strategy works well for small topologies, but visualisations of large topologies often become difficult to read because edge overlaps can occur and switch branches are not correctly mapped. It may be helpful to vary the settings slightly to achieve better results. Nevertheless, the visualisations can help to better understand the topology if no other visualisation data is available.

Listing 21: Use of `linearElementProjection` in the Advanced Example

```
<linearElementProjection id="vis01_lep24" refersToElement="ne64">
    <coordinate x="3747.730" y="782.273"/>
    <coordinate x="3630.000" y="900.000"/>
    <coordinate x="3450.000" y="900.000"/>
</linearElementProjection>
```

Listing 22: Use of `spotElementProjection` in the Advanced Example

```
<spotElementProjection id="vis01_sep097" refersToElement="swi31">
    <coordinate x="-990.000" y="1050.000"/>
</spotElementProjection>
```

Typically, railML files with detailed topologies should contain explicit visualisation data, so it makes more sense to use them prioritised. For this purpose, railML 3 has its own schema `visualization`, which mainly contains positional information for the visualisation of a topology together with its infrastructure elements. Applications make use of this schema in different ways. Two of these possibilities are presented by the following custom graph definitions `D4R_customGraph` and `NOR_customGraph`.

8.1.2 Graph Defined by Fixed Positions for All Elements (`D4R_customGraph`)

This visualisation strategy makes use of the elements `spotElementProjection` and `linearElementProjection` which are both part of the `visualization` schema. As the name suggests, `spotElementProjections` project an arbitrary element from the infrastructure schema onto a single coordinate, which is independent of any positioning system defined in `infrastructure` and is intended only for visualisation. The same applies to `linearElementProjections`, with the only difference that several coordinates can be specified, which are connected by straight lines according to their order in the file from top to bottom.

In the Advanced Example created with the D4R Track Planner, the visualisation data for the infrastructure elements is managed by `spotElementProjections` and for `netElements` by `linearElementProjections`. This is used to define a path over several coordinate points for a `netElement` as shown for `ne64` in Listing 21. One coordinate is specified for almost every infrastructure element, as in Listing 22 for the switch `swi31`. For some elements, a position specification is missing and due to the linear projection of the net elements without a connection between coordinates of the visualisation and the intrinsic coordinates of the net element, the position can not be derived from intrinsic coordinates (as is the case for the next strategy).

Listing 23: Use of `spotElementProjection` in a Norwegian Example

```
<spotElementProjection id="sep_ic1" refersToElement="ne_154_ic1">
    <coordinate x="1320.0" y="680.0"/>
</spotElementProjection>
<spotElementProjection id="sep_ic2" refersToElement="ne_154_ic2">
    <coordinate x="1350.0" y="680.0"/>
</spotElementProjection>
```

Listing 24: Use of `associatedPositioningSystem` in a Norwegian Example

```
<netElement id="ne_154">
    <associatedPositioningSystem id="ne_154_aps01">
        <intrinsicCoordinate id="ne_154_ic1" intrinsicCoord="0.0"/>
        <intrinsicCoordinate id="ne_154_ic2" intrinsicCoord="1.0"/>
    </associatedPositioningSystem>
</netElement>
```

To create a graph in Graphviz that uses the specified coordinates, the layout engine *neato* provides the attribute `pos`, which fixes the location of a node to the specified position. For this to work correctly, the coordinate must be provided separated by a comma and followed by an exclamation mark[21]. In the `D4R_customGraph` definition this is done by the following line (simplified):

```
pos:  MU(e_coord'x)^",-"^MU(e_coord'y)^"!"
```

Here the "-" in front of the y-coordinate is responsible for flipping the visualisation horizontally, which is necessary to get the same view as in the D4R Track Planner.

8.1.3 Graph Defined by Fixed Positions for Net Elements (`NOR_customGraph`)

The railML files provided by the Norwegian Railway Directorate (see Section 10.2 for details) contain visualisation data which only specify points for certain intrinsic coordinates of net elements. As shown in Listing 23, there exists a `coordinate` for each intrinsic coordinate of a net element. This refers to the corresponding `intrinsicCoordinate` of an `associatedPositioningSystem` associated with a `netElement` as it can be seen in Listing 24. It is possible to specify positions of multiple intrinsic coordinates. For correct visualisation it is essential that at least the positions of the bounding intrinsic coordinates 0.0 and 1.0 are specified. Otherwise, the positions of infrastructure elements on the net element cannot be correctly inferred.

With correctly specified positions, the visualisation of the net elements can be done by adding a virtual node for each intrinsic coordinate and connecting these nodes by edges

[21] https://graphviz.org/docs/attrs/pos (accessed 29/10/2023)

labelled with the identifier of the corresponding net element. In addition, this way of storing positions has the advantage that any position of infrastructure elements can be inferred from their intrinsic coordinates. As there are multiple intrinsic coordinates allowed, the position must be computed for the corresponding section of the intrinsic coordinate. This leads to the following general formula for computing the coordinate in the visualisation of an element at intrinsic coordinate $iC_{element}$:

$$(x_1 + \frac{iC_{element} - iC_1}{iC_2 - iC_1} \cdot (x_2 - x_1),\ y_1 + \frac{iC_{element} - iC_1}{iC_2 - iC_1} \cdot (y_2 - y_1)),$$

where iC_1 and iC_2 are the intrinsic coordinates that enclose $iC_{element}$ and (x_1, y_1) and (x_2, y_2) are the coordinates of the `spotElementProjection` of iC_1 and iC_2. Note that $iC_{element}$ can take any value in $[iC_1, iC_2]$, while iC_1 and iC_2 must be intrinsic coordinates specified in a associated positioning system of the net element[22]. For example, a signal at spot location (`ne_154`, 0.4) regarding the example in Listings 23 and 24 would be located at

$$(1320 + \frac{0.4 - 0}{1 - 0} \cdot (1350 - 1320),\ 680 + \frac{0.4 - 0}{1 - 0} \cdot (680 - 680)) = (1332.0, 680.0).$$

The positions of the nodes in Graphviz are fixed by using neato in the same way as described in the previous Section 8.1.2. An example for a visualisation using `NOR_customGraph` can be found in Figure 16a in Section 10.2.

8.2 Second Step: Convert SVG for Use in VisB

The main problem with the SVG output by Graphviz for use in VisB is that the specified IDs are set for the group element containing all elements of a particular node or edge. Because of this, attributes of the inner elements, such as the filling of a node's `ellipse`, cannot be accessed by VisB. It is therefore necessary to generate additional IDs for the members of each group. Since for the visualisations created here it can be assumed that each element does not occur more than once per group, a naive approach of creating IDs based on element types is appropriate. For an ellipse in a group with the ID `ex1` this results in the ID `ex1_ellipse`.

For the `D4R_CustomGraph` there is the special case, that the paths are formed by a linear sequence of points, which results in having multiple path elements per group. As this would break the above described approach for addition of new IDs, these paths are merged into one single path element, which results in an equivalent visualisation.

In order to be able to display the state of the tracks in VisB more easily, the paths are multiplied over each other for all definitions. Thus there are several paths with the same trajectory, whereby these represent the occupancy, route reservations, overlaps, and

[22]Currently, all intrinsic coordinates are interpreted as belonging to one positioning system.

Listing 25: Example of a Linear Gradient in Converted SVG

```
<linearGradient gradientUnits="userSpaceOnUse" id="ne1_lg_occ"
    x1="1.0" x2="2.0" y1="3.0" y2="4.0">
     <stop id="ne1_lg_occ_1" offset="20%" style="stop-opacity:0"/>
     <stop id="ne1_lg_occ_2" offset="20%" style="stop-color:red;
        stop-opacity:1"/>
     <stop id="ne1_lg_occ_3" offset="50%" style="stop-color:red;
        stop-opacity:1"/>
     <stop id="ne1_lg_occ_4" offset="50%" style="stop-opacity:0"/>
</linearGradient>
```

reserved TVD sections. For some of the paths, an `animation` property is added that allows it to blink. To display properties partially on net elements, `linearGradients` are also added for each path. These gradients allow to control the colouring of a path in percentages by `stop` elements, for example transparent from 0% to 20%, red from 20% to 50% and transparent again from 50% to 100% to represent a train position from intrinsic coordinates 0.2 to 0.5 on a net element. This can be encoded in the generated linear gradient of the path visualising the occupation as shown in Listing 25, where the default is 0% for the first two stops and 100% for the second two stops (i.e. the path is transparent).

Initially, the conversion was done by a B machine, which adds the additional IDs, and a Python script that performs the path adoptions. However, this is a very cumbersome procedure and not user-friendly. With the integration of the import process into ProB2-UI, the SVG conversion was revised and transferred into Java code. The creation is now possible automatically during the loading of a railML file without manual steps. In addition, there are several configuration options that are described in more detail in Section 9. For conversion, the SVG is treated as a plain XML file, which is parsed by a DOM parser (`org.w3c.dom.Document`). Then the DOM of the SVG is manipulated for use in VisB as explained above. Finally, the manipulated DOM is written back to the original file (i.e. the conversion is carried out in-place). For each SVG type there are additional tests available in `RailMLSvgConverterTest` using `XMLUnit` to check the correctness of the conversion.

8.3 Visualisation in VisB

The converted SVG can be included in the animation machine by using the definition `VISB_SVG_FILE`, which must contain the path to the generated SVG file. For control of the SVG elements, the VisB definitions introduced in ProB 1.12.0 and 1.12.2 (events) are used in the definition file `RailML3_VisB.def`. They provide convenient access to properties of elements via sets of identifiers, which would not be possible via items in a VisB JSON file. Such a configuration file was created by another Python script when the definitions for creating VisB events were not yet available. This step can now be omitted by the new definitions and is no longer maintained.

There are four types of VisB definitions that are used. The `VISB_SVG_UPDATES` are for the actual visualisation of the current state properties. This is done by providing a set of records with entries corresponding to the SVG attributes to be affected. In addition, each record must contain a field `‘id‘` for addressing the desired element. This is also the reason why the previous conversion of Section 8.2 is necessary. A simple example of such an update definition is shown in Listing 26.

Listing 26: VisB Update Definition for Visualisation of Signal States

```
VISB_SVG_UPDATES16 ==
  {i_e • i_e : RailML3_IL_SIGNAL_CONTROLLED |
    rec(‘id‘: i_e^"_ellipse", stroke: //[...],
      fill: IF aspect_closed : IL_signal_states(i_e) THEN "red"
        ELSIF aspect_caution : IL_signal_states(i_e) THEN "yellow"
        ELSE "green" END ) };
```

It controls the appearance of the stroke and fill of the ellipse representing a signal contained in `RailML3_IL_SIGNAL_CONTROLLED` according to its state (i.e. aspect). All visualised properties can be obtained from Table 2. Figure 12 shows a selection of these properties visualised in VisB. Also via VisB updates, tooltips are provided with additional information on some elements, which can be displayed by hovering over such an element. This includes the detailed state of crossings, derailers, and switches, the current aspects of signals, but also the IDs of trains that are on a net element or a TVD section.

Furthermore, VisB also enables the interactive triggering of events via elements of the visualisation. This link is established by the `VISB_SVG_EVENTS` definitions, which are implemented for events of movable elements and changing of signals. Listing 27 illustrates a simple example for derailers.

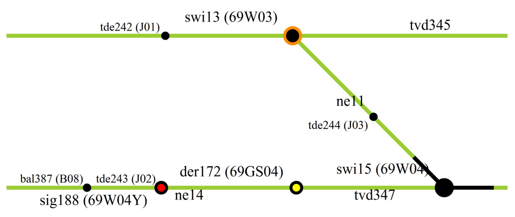
Listing 27: VisB Event Definition for Movement of Derailers

```
VISB_SVG_EVENTS21 == {x • x : allIdsOfType("derailerIS") |
    rec(‘id‘: x, event: "IS_startChangeDerailer",
        predicate: ("DerailerId=","\""^x^"\"") ) };
```

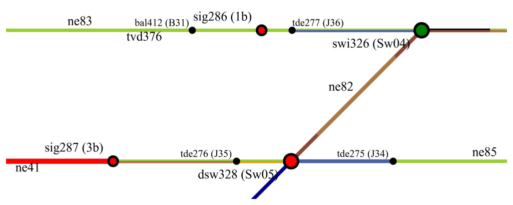
It addresses the SVG group containing all elements of the derailer by its ID and uses the additional predicate to specify that `IS_startChangeDerailer` should be executed (if enabled) for the associated `DerailerId` when the user performs a click on one of the group's elements. A current limitation is that events and hovers can only be specified using constants, so `allIdsOfType` must be used for the rule-based animation machine. In the generated machines (Section 6), the railML specific variables are converted to constants which allow a more precise formulation of these definitions (e.g. `RailML3_IL_SIGNAL_CONTROLLED` instead of `allIdsOfType("signalIS")`).

Table 2: State Properties Visualised in VisB

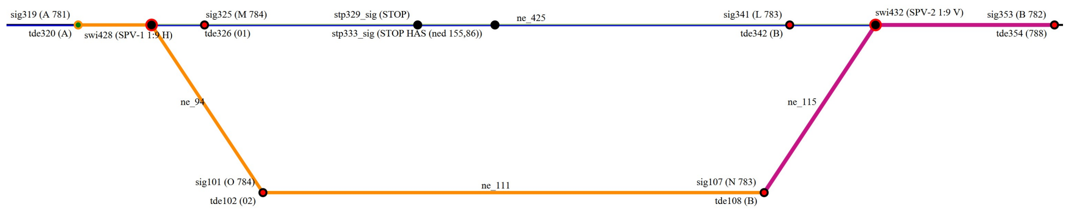
Object	Affected Attribute	Default	Condition	Effect
`derailerIS`	`stroke`	`black`	is locked	`red`
	`fill`	`black`	in derailing position	`yellow`
`movable-Crossing`	`stroke`	`black`	is locked	`red`
	`fill`	`black`	related `route` is in reservation and position matches route	`green`
		`black`	related `route` is in reservation & position does not match route	`red`
`netElement`	`stroke` (free)	`green`	part of passable crossing/ switch branch (and `IS_next`)	`black`
	`stroke` (tvd)	`transparent`	related `tvdSection` is occupied	`blue`
		not blinking	related `tvdSection` is part of route in reservation	blinking
	`stroke` (res)	`transparent`	related `route` is in reservation or reserved	`orange`
		not blinking	related `route` is in reservation	blinking
	`stroke` (ovl)	`transparent`	related `overlap` is in reservation or reserved	`violet`
		not blinking	related `overlap` is in reservation	blinking
	`stroke` (occ)	`transparent`	occupied by a train	`red`
`signalIS`	`stroke`	`black`	signal is locked by a signal plan	`orange`
	`fill`	`red`	shows proceed aspect	`green`
`switchIS`	`stroke`	`black`	is locked	`red`
	`fill`	`black`	related `route` is in reservation and position matches route	`green`
		`black`	related `route` is in reservation & position does not match route	`red`



(a) Switches (in Movement, in End Position), Derailer in Derailing Position, and Closed Signal



(b) Train (Red) Starts Route Reservation



(c) Reserved Route with Overlap and Passable Signal

Figure 12: Visualisation in VisB

Listing 28: VisB Hover and Object Definition for Crossings

```
VISB_SVG_HOVERS3 == {x • x:allIdsOfType("crossing") |
    rec(‘id‘: x^"_ellipse", ‘stroke-width‘: "1.5" )};
VISB_SVG_OBJECTS3 == {x • x:allIdsOfType("crossing") |
    rec(‘id‘: x^"_ellipse", svg_class: "ellipse",
        ‘stroke-width‘: "1.0" )};
```

Finally, hovers are used to indicate elements that are triggers for events. The desired elements and their default appearance are declared in `VISB_SVG_OBJECTS` and the hover behaviour in `VISB_SVG_HOVERS`, as illustrated by Listing 28.

Limitations of the `linearGradient` Approach Since VisB can only address elements by their ID, predefined stop elements with ID and colour are required for each linear gradient. Consequently, only one inner colour section can be predefined for each net element, which is done here by two stops for the outer colour and two for the inner colour. Depending on the state of the net element, the percentage can only be set to one value, which is selected as the minimum or the maximum of, for example, the occupied part of a net element. For an accurate visualisation, the number of occupying trains would have to be known in advance in order to define stops for each train, which is unfortunately not possible. This results in the restriction that only one train, one occupied TVD section, one overlap and one route can be correctly visualised per path object (hence the paths of each property are created separately for a net element). However, this should rarely be a problem as the net elements are the smallest components of the network and therefore there are few cases where multiple routes or trains can occupy the same net element. Another problem is that using `userSpaceOnUse` (cf. Listing 25) only draws a straight line from the start of the path to its end, without considering its curvature. This can lead to inaccurate visualisations if a net element is very curved. The problem should also be rare, as most of the net elements are drawn as straight lines, where the problem does not apply. An alternative approach is to use small blocks that cover the entire route network [Gru+23] and change colours according to their state. While this may be easier for VisB to handle, it would require more effort to convert and generate the blocks, which is why the linear gradient approach was chosen for the current visualisation.

9 Integration in ProB2-UI

As the native application of the conversion process via the print machines (Section 6) and manual creation of visualisations might be too cumbersome for domain experts who are not familiar with the B-method, the idea of developing a user interface for the convenient import of a railML file arose. Since a JavaFX-based graphical user interface to ProB already exists with the ProB2-UI [Ben+21], it was decided to integrate the functionality either directly into the ProB2-UI or as a plugin using the plugin mechanism by Heinzen [Hei18]. As the plugin presented here is currently still under development and the future of the plugin mechanism in ProB2-UI is subject of current discussions, this decision has not yet been finalised.

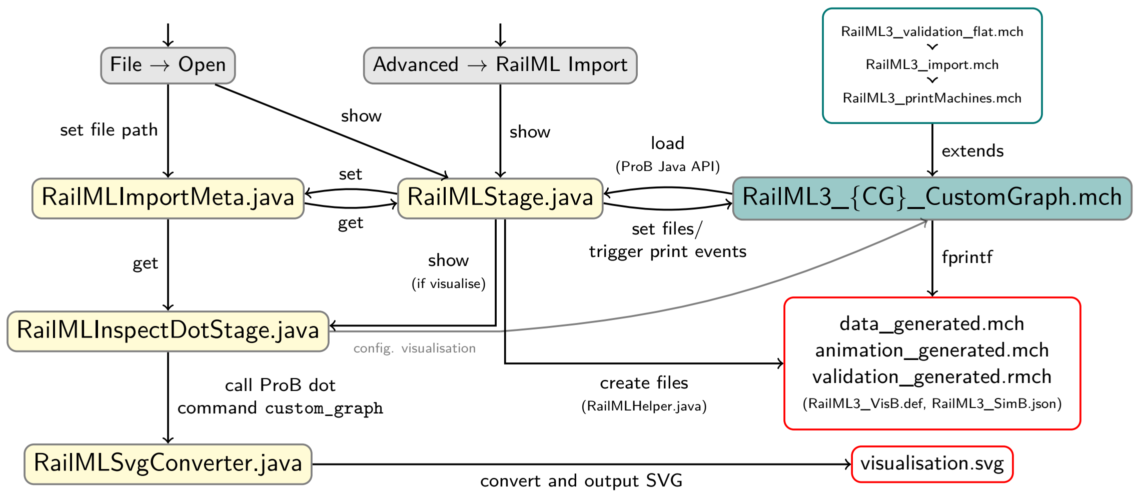


Figure 13: Architecture of the ProB2-UI Plugin

The current architecture of the plugin is outlined in Figure 13. Its core idea is to load the existing machines for import and print using the ProB Java API [Kör+21] to generate the corresponding machines and at the same time generate an SVG for VisB using Graphviz. The required machines (top right box) are written so that they can be used without modification either for the plugin or directly via an extending machine (Section 6). In addition, there are three machines `RailML3_{CG}_CustomGraph.mch` dedicated for the plugin, one per custom graph definition `CG`, which extend the print machine `RailML3_printMachines.mch`. This allows to use the custom graph definition based on the user's choice, which currently cannot be set directly when loading a machine via the API. The constants that would be set via an extending machine (e.g. file names and paths) are passed here as additional predicates when calling `$setup_constants` via the API.

The user interface of the plugin comes with two additional stages, namely the *RailMLStage* for selection of options and file locations before the import process and the *RailMLInspect-DotStage*, which allows the user to configure the visualisation that will be generated by Graphviz. A separate Java class *RailMLImportMeta* maintains the metadata that needs to be exchanged between stages, such as the current state of the loaded machine, the file path for the generated files, and the selected visualisation strategy. The import process can be triggered either by directly opening a railML file from the *File* menu, or from a separate menu item in the *Advanced* menu. In both cases the RailMLStage will be displayed (Figure 14a). If the import was started by opening a file, the file path is already set in the metadata and will be loaded from there, otherwise the file has to be selected first. The user must then select a location for the generated files. If none is selected, the files are placed in the same directory as the railML file.

After file selection, it can be selected if animation, validation machine, and a visualisation should be generated. If one of the machines is selected, the data machine is also generated. For the animation machine, the SimB configuration file and, if a visualisation is selected, the VisB definition file are also copied to the output directory. For the visualisation, the user has to choose one of the supported strategies (see Section 8.1). This affects which custom graph definition is used and therefore which associated plugin-machine is loaded. Depending on the selection made by the user, the generated files are displayed in a list below.

By starting the import, a `RailML3_{CG}_CustomGraph.mch` machine based on the selected visualisation strategy *CG* is loaded via the ProB Java API. The required machines are stored as resources in the project. Since the ProB Java API requires a path as a string to load a machine, problems occur when ProB2-UI is started from a JAR file, where the file system of the resources is no longer accessible. To avoid this, in this case all resource machines are copied as temporary files before loading and deleted immediately afterwards.

For the import, a new state space is created (independent of the ProB2-UI animator), as the machine should only be loaded in the background, using a separate thread, for the import and not animated. The machine `RailML3_printMachines` leaves several constants unspecified that can be used to control the file and machine names. These are specified by additional predicates that are passed with the execution of `$setup_constants`:

```
State currentState = stateSpace.getRoot()
  .perform("$setup_constants", "file = \"" + railMLpath + "\"" + ...)
  .perform("$initialise_machine");
```

To improve performance when only the visualisation is to be generated, there is also the constant `FULL_IMPORT`, which allows the import to be aborted early when all the data required for the visualisation has been loaded. After initialisation it is checked whether there are invariant violations or other import errors (then `no_error = FALSE` applies). If not, the printing operations can be performed. Just before this, the necessary files are created on Java side by the function *replaceOldFile* in the *RailMLHelper* class, as the ProB `FPRINTF` function can only deal reliably with existing files. If a machine file with the same name already exists, it is copied with an ascending number, and the original file is emptied (`FPRINTF` would just append the output to the existing content).

Then, the two associated print operations are performed (for all selected machines):

```
replaceOldFile(generationPath.resolve(dataFileName.getValue()));
currentState.perform("triggerPrintData").perform("printDataMachine");
```

Finally, all machines are added to the current project together with the VisB definition file (depending on the selection) and the SimB configuration file is also linked to SimB.

After the import of the railML file and (if selected) generation of machine files, the ProB dot command `custom_graph` is executed as *DotVisualizationCommand* and the resulting visualisation is displayed in the *RailML3_InspectDotStage* as shown in Figure 14b. This stage is based on the source code of the *DotView* for all possible graph visualisations in ProB2-UI, but extends it with possibilities to configure the railML specific visualisation. The options available vary according to the visualisation strategy selected. For the `DOT` strategy, it is only possible to select whether to display names and in which language (currently English, Norwegian, and German), as the other elements are mandatory for correct visualisation. In addition, it is possible to choose from three dot engines (dot, neato, fdp) and whether curved splines are to be used. For the other two strategies (`D4R`, `RAIL_OSCOPE`), all displayed elements can be selected and deselected (e.g. balises, level crossings, signals) and the scaling can be adjusted to improve the visualisation when many elements are close together. Internally, a listener is set for each option which, when clicked, performs the associated operation in `RailML3_{CG}_CustomGraph.mch`:

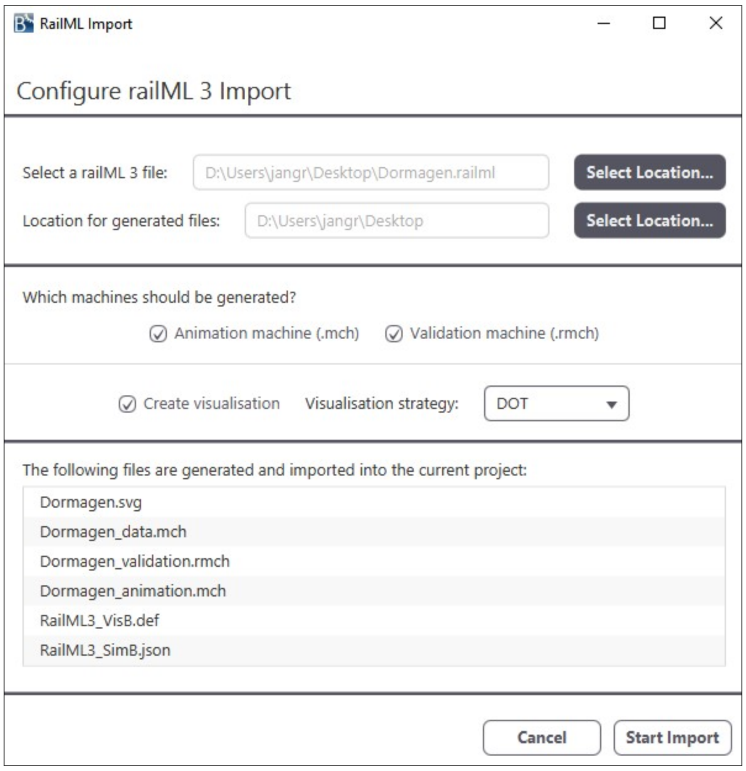
```
balises.selectedProperty().addListener((o,f,t) ->
    railMLImportMeta.perform("changeDisplayBalises"));
```

For example, if the user does not want balises to be displayed, the operation

```
changeDisplayBalises =
   ANY b WHERE b /= DISPLAY_BALISES THEN DISPLAY_BALISES := b END;
```

is performed for the shared state in *RailMLImportMeta*. The custom graph definitions recognise the boolean variable `DISPLAY_BALISES` and will replace the set of balise IDs with the empty set (making them disappear). After each change, the visualisation can be reloaded to apply the changes and also saved. The latter only saves the SVG created by Graphviz, which has not yet been converted for VisB.

The conversion of the SVG for VisB (Section 8.2) starts as soon as the user is satisfied with the current visualisation and accepts it with the apply button. For this purpose, the *RailMLSvgConverter* class provides the necessary static methods. As mentioned in Section 8.2, the conversion is done in place, so the SVG generated by Graphviz is first saved to a temporary file, to which the conversion is then applied. Afterwards the resulting file is copied to the directory of generated files.



Configure railML 3 Import

Select a railML 3 file: D:\Users\jangr\Desktop\Dormagen.railml [Select Location...]

Location for generated files: D:\Users\jangr\Desktop [Select Location...]

Which machines should be generated?

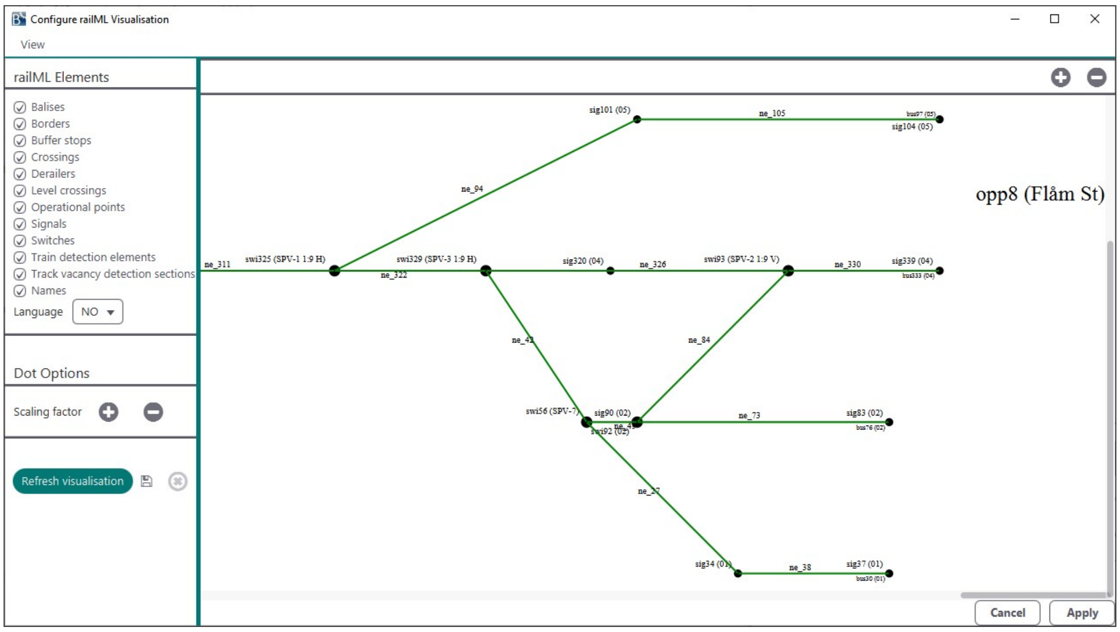
✓ Animation machine (.mch) ✓ Validation machine (.rmch)

✓ Create visualisation Visualisation strategy: DOT

The following files are generated and imported into the current project:

- Dormagen.svg
- Dormagen_data.mch
- Dormagen_validation.rmch
- Dormagen_animation.mch
- RailML3_VisB.def
- RailML3_SimB.json

(a) Configuration of the Import



railML Elements

- ✓ Balises
- ✓ Borders
- ✓ Buffer stops
- ✓ Crossings
- ✓ Derailers
- ✓ Level crossings
- ✓ Operational points
- ✓ Signals
- ✓ Switches
- ✓ Train detection elements
- ✓ Track vacancy detection sections
- ✓ Names

Language: NO

Dot Options

Scaling factor

(b) Configuration of the Visualisation

Figure 14: RailML Import in ProB2-UI

This completes the import process. If machines have been selected for generation, they are added to the current project and the animation machine is automatically loaded (validation machine if animation was not selected). For the validation machine it is recommended to install the “Rule Validation Language Plugin” in the ProB2-UI for convenient validation of custom rules (Section 6.2).

For future development, other features such as better display and export of validation messages (possibly together with a general revision of error messages in the UI), visualisation of validation errors in VisB, and the creation of different scenarios during generation (initial train positions) are possible.

10 Case Studies

A crucial problem in developing applications for the relatively new railML 3 format is the lack of publicly available example files. The only official example is the so-called “Simple Example” from railML.org, which only demonstrates the functionality for a very basic topology. The so-called “Advanced Example” is still under development and was kindly provided by railML.org for use in this project. Fortunately, the Norwegian railway data is available as the only real world example in railML 3.2 in a development state [Jer23]. Further own examples have been created using the D4R Track Planner software (see Appendix B). However, these are only semi-real as they are based on track plans from an interlocking simulator.

This section presents selected models to which the previously presented techniques have been applied. The results are then discussed for each model. In general, errors have already been successfully detected in some of the models.

10.1 Examples by RailML.org

Since the examples provided by railML.org[23] can be assumed to be used in accordance with the intent of the schema, some implementation details have been adapted to these examples. Occasionally, this also supports cases that are not directly covered by the schema documentation.

Simple Example The Simple Example is the smallest example available. It consists of a single track line between two stations with two tracks each, as shown in Figure 15. Except for the level crossing, all the infrastructure elements shown in yellow can be imported into ProB. In total, the model consists of 12 net elements, 13 net relations, 3 switches, 1 derailer, 0 crossings, 13 signals, 13 TVD sections, 3 routes, 2 overlaps, 8 route release groups, and 0 open ends. As the only example, it implements 2 conflicting routes, 2 route relations, and signalling using 2 aspect relations.

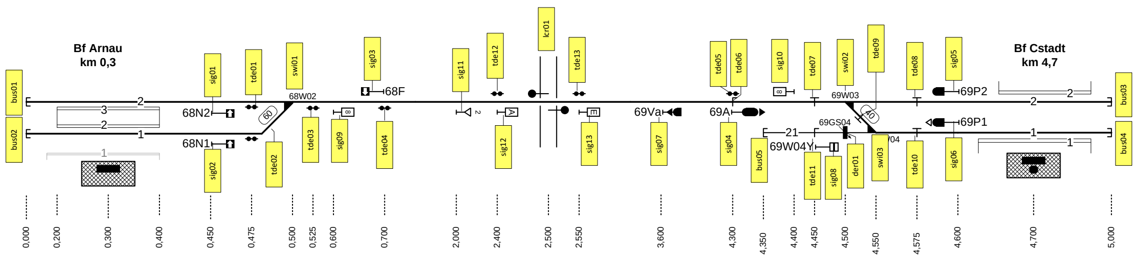


Figure 15: Simple Example for railML 3.1 provided by railML.org [Rai23a]

[23]https://www.railml.org/en/user/exampledata.html (accessed on 20/10/2023)

To investigate the example, it is imported using the plugin for the ProB2-UI (Section 9). The recent published version 12 of the example did not pass the semantic validation due to a minor rule violation:

```
Error: RailML: [Line 952, id 'pt_swi02']: Switch should return
       to preferred position, but no preferred position is defined
```

As the switch was adapted to not return to its preferred position, the model passed both syntactic and semantic validation without error. However, semantic warnings informed about the detection of three overlapping TVD sections, namely `X01T`, `LX2.5T`, and `X02T`. It turned out that this is due to a misconfiguration of the `spotLocation` of `tde13`, which lacks the `pos` attribute and is therefore placed at the default intrinsic coordinate 0.0 of `ne_x01`. The train detection element `tde12` has the (inferred) location (`ne_x01`, 0.472). But according to Figure 15, `tde12` should be to the left of `tde13` (the normal direction is to the right in the figure), which is not fulfilled by the current specification. Adding a proper position to the `spotLocation` should fix this issue. The remaining warnings are for the inferred intrinsic coordinates from the `pos` attributes.

Since no visualisation data is provided, only the *DOT* strategy is applied, resulting in the visualisation shown in Figure 11. It captures the structure of the example quite well, but suffers from the limitations of the `DOT` strategy in terms of visualising signal states and train detection elements.

The above issue regarding TDE locations is also implicitly detected by the example rule `MinDistanceOfTDEs` in the generated validation machine by the following counter example:

```
"Distance between trainDetectionElements tde04 and tde13 is 0.0 (<= 21.0)"
```

This shows that both `tde04` and `tde13` are located at the same point in the topology, which is indeed not correct. The other example rule about switch speeds can be successfully validated.

The animation model was then simulated with SimB once in real-time and 20 times with a Monte Carlo simulation using the hypothesis test for the fulfilment of all invariants for traces with 300 steps. For this, two trains are initially placed on `ne_a01` and `ne_b05`, as the topology has no open ends for arrival of trains. In the Monte Carlo simulation, none of the 20 executions were successful, i.e. an invariant violation occurred in every trace. This is due to the placement of the second train on `ne_b05`, which from there overruns the derailer `der01` in derailing position and partly also the wrongly placed switch `swi03`. The reason is that the signal `sig08` protecting these elements was not recognised as a controllable signal during import because it has no associated interlocking element. Whether this is intentional or an error must subsequently be assessed by domain experts. Also, after changing direction before signal `sig06`, it may enter the set route of the other train going from `sig04` to `bus03`, resulting in a train collision. Here it would have to be ensured that the train can only change its direction behind the signal so that the return journey is secured by it. However, these are also questions of modelling behaviour in B, which can be adapted if necessary. As a further restriction, only routes from `sig01` and `sig02` via `sig04` to `bus03` are currently implemented, so that the first train can never reach the track in the direction of `bus04`.

Furthermore, partial model checking with deadlock checking was applied and stopped after 60 % of the current queue had been processed (432 816/720 012 states). Until then, eight deadlocks with invariant violations were found, which occurred for the same reason as in the simulations. A run with enabled invariant check revealed that the route relations cannot be fulfilled with the current modelling. These require for the two routes leading to `sig04` that the activation section of the routes must be continuously in an occupied state. However, as the train will eventually leave the section while using the route, the violation is unavoidable. It could either be that the property is only intended to be checked at the time of reservation of the route, or that the reservation of route blocks is also meant by occupancy. In this case the modelling in B would have to be adapted.

In general, even this small example clearly suffers from the state space explosion problem, but with the help of running a few simulations, some errors could still be discovered.

Advanced Example This example is a slightly larger topology with a total length of 25 kilometres and a greater variety of movable element types. Here the file created with the D4R Track Planner and thankfully provided by railML.org is examined, there is another one available in railOscope. The model comes with 74 net elements, 107 net relations, 21 switches, 1 derailer, 1 (movable) crossing, 31 signals, 36 TVD sections, 38 routes, 0 overlaps, 0 route release groups, and 1 open end. A special feature that does not correspond to the schema is that the branches of the interlocking objects of switches are indicated by net elements instead of tracks. Therefore, this is currently also supported by the model. In fact, the track data is internally projected onto the net elements anyway. This model also lacks length specifications for net elements, which means that the intrinsic coordinates cannot be correctly derived for given inner positions. Warnings are issued about this during import. The validation of the model ends without errors, only some semantic warnings regarding the TVD sections are generated, for instance:

```
Warning: RailML: [Line 653, id ’ne88’]:
         netElement is not part of any tvdSection.
```

Some routes also have TVD sections that are behind the exit signal. As these are more likely to be part of an overlap, a warning is created.

Both *DOT* and *D4R* strategies can be used for visualisation. However, it is recommended to use the available data with the *D4R* strategy, as the network is a bit too twisted for the *DOT* strategy.

One real-time simulation and Monte Carlo simulations were again carried out to investigate the behaviour. Seven out of twenty runs were successful (i.e. without invariant violation). For the remaining unsuccessful traces, the track at the open end was found to be the cause. As this track has no signal protection, arriving trains can run over the switches `swi45` and `swi49` if they are not set correctly, and also enter reserved routes on the main line, which can lead to train collisions. The trace of the real-time simulation represents a successful run where all trains have reached the final position. It has also been exported as a standalone HTML file, allowing playback without any knowledge of ProB, for easy inspection by a domain expert [VHL22].

10.2 Norwegian Railway Network

The entire Norwegian railway network operated by Bane Nor has been made publicly available by the Jernbanedirektoratet in both railML 2 and railML 3 [Jer23], while work on railML 3 is still ongoing, according to Brand [Bra23]. As this project focused on railML 3, only the not completed data in railML 3.2 could be used for testing. The files contain detailed infrastructure and basic interlocking data with the topologies varying considerably in size. There are small lines, such as Flåmsbana, with only a few kilometres and a few branches, long lines with several hundred kilometres, such as Bergensbanen, and spatially small topologies with many branches, such as Oslo Sentralstasjon.

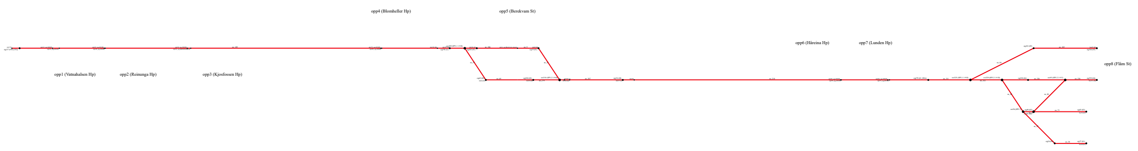
The models are created with railOscope, which then generates a railML 3.2 file from the data. With the help of semantic validation, some errors could be detected in the exported models (e.g. for the former version of `DOB1.xml`). On the one hand, the entry and exit signals were identical for some routes, which does not form a correct route, and on the other hand, some route paths could not be computed completely because contradictory switch positions were specified:

```
Error:  RailML: [Line 15761, id 'rte1566']:  Forced switch position
        for switch 'swi1044' does not match the route path.
```

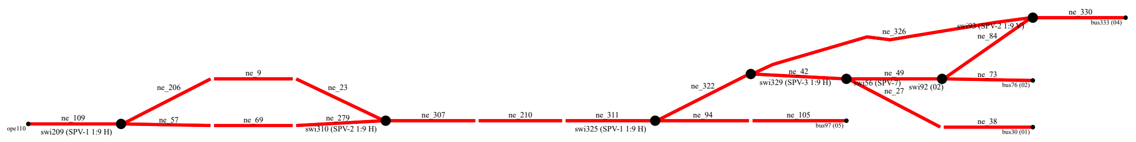
The investigation revealed that the branches of all switch crossings on the interlocking side had been swapped during export, so that the forced switch positions did not match the branches of the intended route. This bug has been reported and fixed, so there are now "newer" Norwegian models that mostly pass the semantic validation. In addition, all crossings (including non-movable ones) are supplemented with `switchIL` elements in the models. As it is not clear what behaviour this should specify, crossings defined in this way are currently treated as static. Again, some additional properties have been implemented that do not actually correspond to the schema, such as allowing switches to be used in addition to detection elements for `isLimitedBy` of an overlap.

Due to the complexity of the models, there is also an explosion of the state space, and even Monte Carlo simulations are currently hardly applicable due to the performance of the models. Hence, their behaviour has mainly been investigated using real-time simulations.

Flåmsbana (FLB) This is one of the simplest models of the "old" dataset and consists of 25 net elements, 32 net relations, 7 switches, 0 derailers, 0 crossings, 29 signals, 10 TVD sections, 14 routes, 3 overlaps, 5 route release groups, and 1 open end. The specified line has a total length of about 20 kilometres[24]. The model can be successfully imported without errors and visualised using the given visualisation data (Figure 16a). The visualisation without using the existing data shown in Figure 16b also captures the topology very well.



(a) Using `NOR_customGraph`



(b) Using `DOT_customGraph`

Figure 16: Visualisation of Flåmsbana

Using real-time simulations involving two trains, at least two erroneous traces with invariant violations were found. The first issue is, as in the Advanced Example, that the track is not protected by a signal for incoming trains until the first route at Berekvam can be reserved. This allows the two trains to enter the interlocking area one after the other and collide before the first signal. The second problem could be identified as an incomplete configuration of the last TVD section in Flåm. Because `bus333` is not specified as one of its demarcating elements, this track is not part of the section, resulting in an occupied section that is no longer used by a train. This means that the reserved route cannot be released and cannot be reserved by the second train waiting in front of Berekvam.

Bergensbanen (BB) This model is particularly interesting for assessing performance, as it is the largest model in terms of distance, with a total distance of 371 kilometres[25]. This is also reflected in the number of elements: 353 net elements, 543 net relations, 132 switches, 0 derailers, 1 crossing, 422 signals, 375 TVD sections, 372 routes, 120 overlaps, 298 route release groups, and 1 open end. The model was successfully validated and visualised, and no errors occurred during two real-time simulations, which are time-consuming due to the size of the model.

[24]`https://en.wikipedia.org/wiki/Fl%C3%A5m_Line` (accessed on 22/10/2023)
[25]`https://en.wikipedia.org/wiki/Bergen_Line` (accessed on 22/10/2023)

Oslo Sentralstasjon (OSL) The "Sentralstasjon" (main station) of Oslo is a comparatively large station with 19 platforms[26] and therefore is an interesting example of a complex topology with many switches. With Bergensbanen, it is also one of the largest models available, which makes it interesting for the performance benchmarks in Section 11. The specification includes 383 net elements, 696 net relations, 203 switches, 0 derailers, 10 crossings, 290 signals, 378 TVD sections, 304 routes, 2 overlaps, 134 route release groups, and 13 open ends. The semantic validation uncovered a misconfiguration of two crossings:

```
Error: RailML: [Line 20765, id 'cro1606']: Turning branch end is
       not connected to straight branch end,
```

which could easily be fixed by swapping the erroneous turning and straight branch. Visualisation with `NOR_customGraph` is possible without restriction, but for `DOT_customGraph` there are too many branches, so that the visualisation is rather incomprehensible. As with Bergensbanen, the simulation takes quite a long time, so only a few test runs of six trains were made in real-time, all of which ran without error.

Independently of these case studies, minor inconsistencies were also found in the other Norwegian models using semantic validation (e.g. missing branches for switches, references to unspecified elements).

[26] `https://en.wikipedia.org/wiki/Oslo_Central_Station` (accessed on 22/10/2023)

11 Performance Analysis

Throughout the development, efforts have been made to improve the performance of the import process. This chapter evaluates different empirical benchmarks for the case studies presented in Section 10. All benchmarks are executed on a Windows 10 Pro 64-Bit (22H2, 19045.3570) with 3.5 GHz Quad-Core Intel Core i5-6600K and 16 GB RAM based on the submitted code (Appendix A). The benchmarks were run using the `ExecuteAndMedian.sh` script, which expects a ProB command with the `--profile` flag, runs the command once without measuring to create the `.prob` file, and then runs the command ten times in succession. All outputs are stored in separate files and evaluated at the end. The median is calculated for the total runtime and walltime and saved together with all individual measurements in `result.txt`. Of particular interest is the walltime, as this is the time that the user experiences at the end. The models evaluated are assumed to be syntactically and semantically valid, so that all import operations are fully executed.

First, the import times for the rule-based machine and the import machine are compared for the different models. Furthermore, the loading times of the generated machines with different preferences are examined. Finally, the benefits from memoization are measured and discussed.

11.1 Comparison of Import Times

In order to be able to import railML data into ordinary B machines, the import machine has been created (Section 3.6), which performs the import operations of the rules machines sequentially. The same can be achieved directly with the topmost rules machine and random animation for as many steps as the number of operations (or execution of all rules in the ProB2-UI plugin for rule validation). From a performance point of view, it is interesting to investigate whether this has an impact on the duration of the import process. For this, the test script was run for each case study with the ProB calls

```
probcli RailML3_validation.rmch
     -execute_all -p MEMOIZE_FUNCTIONS true --profile
```

for direct execution of the operations in the rules machines (the constant `file` has been adapted for each model), and

```
probcli RailML3_import.mch -p MEMOIZE_FUNCTIONS true
    -property "file = \"[...]/RailML examples/[...]/file.xml\""
    -execute 3 --profile
```

for execution using the import machine. The results are shown in Table 3.

Table 3: Comparison of Import Times for Selected Models (in ms)

Model	`RailML3_validation.rmch`		`RailML3_import.mch`	
	Tot. Runtime	Tot. Walltime	Tot. Runtime	Tot. Walltime
Simple Example	5957	9234	6464	11 158
Advanced Example	9345	12 851	9638	15 833
BB	278 986	315 169	290 101	351 104
FLB	7499	10 856	8471	15 076
OSL	138 859	174 770	145 948	198 374

In general, it is not surprising that the import times are significantly longer for large topologies. But even for the two largest topologies currently available (BB and OSL), the runtimes are satisfactory, with a maximum of less than six minutes for import and validation. It is noticeable that, despite a similar number of elements (Section 10.2), the runtimes of BB and OSL differ significantly. This is most likely due to the larger number of signals in the BB model, as the execution of `set_IL_SIGNAL` takes about a third of the total runtime ($\approx$ 100 sec). To compute the signal sections, it is necessary to check for each reachable signal whether there is another signal on the path between them. As many signals can be reached by one signal on long lines like BB, as opposed to a branched network like OSL, this is likely to increase the runtime. For this reason, importing the entire Norwegian network from a single railML file is not realistic at this stage.

It can also be observed that the use of the import machine leads to a degradation of the performance, which is at most about 36 seconds for BB. An important aspect is probably the additional output performed by the import machine. For every warning rule that fails, a warning is generated that has to be output at the end of the import, which can cause the delays. However, this should be investigated further, and if at some point the ProB parser supports the direct inclusion of rules machines in ordinary B machines, an attempt should be made to use the rules machines directly. Perhaps parts of the ProB2-UI import plugin (Section 9) can be also adapted for this in conjunction with the ProB Java API to run the rules machines natively and then generate B machines directly from them.

Besides these differences, the detailed analysis reveals that only a few operations are responsible for the high costs. These are those that require path computations using the topology relation and its (precomputed) transitive closure, including routes, overlaps, TVD sections, and signal sections. As a result of their poor performance, these operations have often been reworked during development to improve their performance significantly, up to halving their individual runtimes.

Finally, it is worth noting that the XML parsing in ProB is not a significant bottleneck for larger topologies compared to their total import runtime. Parsing and conversion takes about 30 ms for the Simple Example and FLB, 120 ms for the Advanced Example, 1060 ms for BB, and 1020 ms for OSL (times estimated from one sample run only).

11.2 Loading Times of Generated Machines

Part of the motivation for creating standalone machines is to achieve faster loading of the machine by omitting the import process, to avoid having to go through the entire import process after making small changes to the machine. Therefore, the loading times of the generated animation machines were measured until the initialisation was completed with the command

```
probcli "Generated data/[model]/[model]_animation.mch"
    (-p jvm_parser_fastrw true) --init --profile.
```

Since there can be efficiency problems with type checking for generated machines that contain very large sets with elements from enumerated sets, loading is tested with the new ProB preference `optimize_enum_set_elems` in addition to the default preferences. It enables more efficient type checking and pre-compilation for such large sets that use enumerated sets, which is why this preference is enabled by default for the generated animation and validation machines (`SET_PREF_optimize_enum_set_elems == TRUE`).

The benchmarks were run with this preference once using the standard format and once using the *fastrw* representation, a format provided by SICStus Prolog, for the `.prob` file. The latter allows for faster read and write access, which could lead to better loading times. Before each run, the `.prob` file was deleted to ensure that a new one is created with the intended format. The results are listed in Table 4 below (*fastrw* results in brackets).

Table 4: Comparison of Loading Times for Selected Generated Machines (in ms)

Model	Default		`optimize_enum_set_elems`	
	Tot. Runtime	Tot. Walltime	Tot. Runtime	Tot. Walltime
Simple Example	2188	4796	2053 (1953)	4471 (4354)
Advanced Example	3580	6652	3002 (2568)	5979 (5420)
BB	18 859	35 097	10 971 (6804)	23 787 (17 064)
FLB	2960	5886	2460 (2178)	5174 (4874)
OSL	18 128	33 234	10 620 (6717)	24 080 (16 794)

As expected, all loading walltimes are much better than for the rule-based variant (cf. Table 3). A detailed analysis shows that most of the time is taken up by parsing and loading the machine, while initialisation takes comparatively little time. As a result, using the preference for optimisation and *fastrw* leads to a significant improvement, up to halving the walltime for the large models, with almost 18 seconds difference for BB. It can also be generally observed that the magnitude of the times is the same for BB and OSL, in contrast to the machines including the import, where the characteristics of the topology lead to differences. For the simple models, the overall times are also satisfactory.

As mentioned earlier, further improvements in loading can be expected if only the required sets from the data are used for the animation and not all of them have to be loaded (currently this corresponds to 130 unused constants).

Another current performance issue is that loading models in ProB2-UI with VisB can again significantly increase loading times, although the definitions used for VisB have already been improved considerably in terms of performance compared to previous implementations. This part therefore requires further investigation, possibly also in VisB itself.

Concluding based on these results, it is clearly recommended to use the generated machines for investigation of a model due to much shorter loading times and persistence of the models.

11.3 Memoization

When dealing with the imported XML data, some queries occur repeatedly, such as finding all elements of a particular type, finding the children of an element of a particular type, or finding the set of all IDs of a particular type (cf. Section 3.1). Initially, these queries were implemented as definitions, but this turned out to be not performant. Therefore, they were converted into abstract constants, which already enables a better typification. In combination with "memoization", a technique of ProB for storing the results of function applications[27], this leads to a significant improvement in performance. The tests of Tables 3 and 4 were therefore already carried out using this technique. The benefit of memoization is also reflected in the statistics of reused stored values during the import in Table 5.

Table 5: Memoization Statistics of Selected Models (values stored/reused)

Model	`elementsOfType`	`childsOf-ElementType`	`allIdsOfType`	Total Reused
Simple Example	126 / 329	386 / 324	29 / 230	883
Advanced Example	126 / 1947	1975 / 1875	58 / 1802	5624
BB	135 / 18782	18822 / 22068	30 / 8907	49757
FLB	132 / 931	968 / 1201	30 / 443	2575
OSL	133 / 16588	16626 / 18332	30 / 17920	52840

Except for `childsOfElementType`, the amount of reused values significantly exceeds the amount of stored values. For comparison, the tests from Section 11.1 were repeated without memoization (`-p MEMOIZE_FUNCTIONS false`) for the import machine. This gave the following results (total runtime/walltime in ms):

Simple Example: 6944/11 531, Advanced Example: 23 463/29 689,
BB: 1 109 866/1 187 239, FLB: 10 909/17 585, OSL: 1 201 694/1 250 241.

[27] `http://prob.hhu.de/w/index.php?title=Memoization_for_Functions` (accessed on 24/10/2023)

Obviously, without memoization there is an unacceptable increase in both runtime and walltime, up to a factor of seven for OSL, which is why enabling this preference is mandatory for good performance in the current models.

Unfortunately, the approach of memoization is not applicable for the definition `elementOfId`, since the polymorphic properties of definitions are required here. Some improvement should come from "record indexing", which in the latest version of ProB uses the alphabetically first field of a record for fast access to records. As this is always the `Id` field for railML records due to the upper case, this also applies to the `elementOfId` definition, as this only searches for the ID of a record.

12 Conclusion and Outlook

In this work, an approach was developed using the B-Rules DSL integrated in ProB to transform railML 3 data to classical B. In addition to syntactic and semantic validation of the imported data, it is possible to animate and simulate the models either by building directly on the rules machines or by using generated machines, which also allow for validation of custom rules. Three visualisation strategies for the topology have been implemented, one of which works independently of the given visualisation data in railML. The development of a plugin for the ProB2-UI was started, which combines all the work, supplemented by configuration options for the visualisation. Finally, a number of case studies were examined, also from a performance point of view, and errors were successfully found and corrected in some files, in particular from the Norwegian railway network.

It has been shown that the import with the included semantic validation is feasible even for large models in an acceptable time. Loading times were significantly reduced by using generated machines with additional settings. The animation and simulation of large models is still a challenge at the moment, as the computation of guards and VisB definitions can become expensive. Model checking is generally only partially applicable due to the state space explosion problem, but can also help to uncover inconsistencies in the specifications.

A challenge for formal verification is the transformation of railML specifications into formal properties, since railML as an XML-based format is not a strictly formal language. This is partly due to the many optional XML attributes and children whose absence must be compensated for by additional derivations, but also due to some places where the schema definition is not clear enough and can be used ambiguously (e.g. the localisation problem discussed). In particular, when evaluating errors, it can be challenging to determine whether the error is due to the modelling in railML or the implementation in B, making complete formal verification difficult for different railML modelling styles. Therefore, the methods presented here should primarily be seen as additional validation to increase confidence in the correctness of a specification.

Future work should therefore address the problems posed by XML through optional attributes for a strongly typed language such as B, for example through records that can handle also optional fields. Further development of the B-Rules DSL should also be considered. For example, there is currently no information output about successful rules for specific elements, which could be interesting for a validation report. Such a report could contain the identities of all successfully validated objects in addition to the already available counterexamples and integrated as additional output to the rule validation plugin of the ProB2-UI. Also of interest for this work is the inclusion of rules machines in ordinary B machines.

On the railML side, future work could take into account the rolling stock and timetabling schemas. One approach could be to assign each train a fixed path through the controlled area, which is then followed according to the timetable. Of course, the import can generally be extended to include more elements and more rules can be added. In particular, the interlocking schema should be implemented in more detail once examples are available.

Finally, it might also be interesting to translate a model specified in B, maybe even already proven to be correct, to railML.

Appendices

A Source Code

The majority of the source code (machine files, generated data, performance results) is located in the "`B-RailML`" repository available at `https://gitlab.cs.uni-duesseldorf.de/stups/prob/b-railml` with the state of commit

"update readme", 30/10/2023,
`4fef15c0981b0966859a0b3479d6b1aed6d7e64c`.

The railML import integration for the ProB2-UI is included in the `railml-import` branch of its repository available at `https://github.com/hhu-stups/prob2_ui` with the state of commit

"final update of resources", 30/10/2023,
`686726bbcabe9bb8959c7055ed4a4f3d698fca8b`.

For the implementation to work correctly, at least ProB version 1.12.3 as of commit `c81e69e3fde16c38dd95bac1e423d763d72e010b` (27/10/2023) is required.

The adapted Rule Validation Language Plugin for the ProB2-UI can be found at `https://github.com/hhu-stups/prob2_ui_rule_validation_plugin`:

"replace search symbol", 11/8/2023,
`4e79f1e4b6172b016725d942a8dc2eb1b328b0ad`.

To be able to load the plugin into the ProB2-UI, its JAR file must be built using the command `./gradlew jar`.

B Case Study Files

The corresponding files of the five case studies can also be obtained from the B-RailML repository in the "RailML examples" folder. The files used are:

- `Simple Examples by railML.org (IS, IL, V2.3-V3.1)/SimpleExample_v12.xml`
- `Advanced Example by railML.org (IS, IL, V3.1)/AdvancedExample.railml`
- `Jernbanedirektoratet Norge (V3.2) - NEW -/BB.xml`
- `Jernbanedirektoratet Norge (V3.2)/FLB.xml`
- `Jernbanedirektoratet Norge (V3.2) - NEW -/OSL.xml`

The other Norwegian examples and some smaller ones exported from railOscope can also be found there. Under "`Own examples created by D4R`" there are a few more own examples created with the D4R Track Planner, based on track plans of an interlocking simulator[28]. These include Cologne, Dormagen, Düsseldorf, Munich, Neustadt (Weinstraße) and the example topology presented by Gruteser et al. [Gru+23]. As a side note, due to the very small topology, model checking (without invariant checking) could be completed after 73 065 states and 252 836 transitions for the latter.

[28]`https://www.stellwerksim.de` (accessed on 20/10/2023)

C Additional Details on the Imported Data

Table 6: List of Derived Relations

Name	Description
Infrastructure (`RailML3_IS_`):	
`NET_ELEMENT_LENGTHS`	Length of all net elements, −1.0 if not specified
`NET_ELEMENT_INTRINSIC_COORDINATES`	Function returning a function of the associated positioning system IDs and their intrinsic coordinate IDs together with their coordinates for a net element
`NET_ELEMENT_ASSOCIATED_POSITIONING_SYSTEM`	As above, but gives a sorted sequence of intrinsic coordinates for each associated positioning system (mainly for `NOR_customGraph`)
`NET_RELATION`	Relation of locations containing all net relations
`NO_NET_RELATION`	Function of net relations with attribute `navigability="None"` per ID
`NET_RELATION_BY_ID`	Function of navigable net relations per ID
`NET_RESOURCES_MICRO_LEVEL`	Set of all net resources on micro level
`NET_RESOURCES_MESO_LEVEL`	Set of all net resources on meso level
`NET_RESOURCES_MACRO_LEVEL`	Set of all net resources on macro level
`OPENEND_IDS`	IDs of borders with attribute `openEnd="true"`
`CROSSING_BRANCHES`	Total function returning the set of all navigable branch relations (as pair of locations) for a crossing ID
`DERAILER_NOT_PASSABLE`	Total function returning the set of all location pairs intersecting the derailer's location for a derailer ID
`SIGNAL_POSITIONS`	Unique spot location for each signal ID
`SIGNAL_IS_SWITCHABLE`	Boolean, indicating whether signal is switchable per signal ID
`SIGNAL_IS_TRAIN_MOVEMENT_SIGNAL`	Boolean, indicating whether signal is train movement signal per signal ID
`SPEED_SECTIONS`	Relation of speed section IDs with their corresponding location pairs
`VALID_FOR_SPEED_PROFILES`	Function returning the set of valid speed profiles for a given speed section
`SWITCH_IDS`	IDs of switches that are not `switchCrossingParts`
`SWITCH_TYPE`	Switch types of for all switches from the above set
`SWITCH_BRANCHES`	Total function returning the set of all navigable branch relations (as pair of locations) for a switch ID from the above set
`TRACK_ASSOCIATED_NET_ELEMENTS`	`RailML3_IS_LINEAR_LOCATION_ASSOCIATED_NET_ELEMENTS` restricted to track IDs
`TRACKS`	Track function consisting of net elements and their begin/end elements (mainly for `DOT_customGraph`)

TRACK_BEGIN_NET_ELEMENT	Location of the track begin per track ID
TRACK_BEGIN	Infrastructure element of the track begin per track ID
TRACK_END_NET_ELEMENT	Location of the track end per track ID
TRACK_END	Infrastructure element of the track end per track ID
LINEAR_LOCATION_ REFERS_TO	Mapping of linear location IDs to their parent IDs
LINEAR_LOCATION_ ASSOCIATED_NET_ELEMENTS	For each element with a linear location: its plain relation specified by associated net elements
LINEAR_LOCATIONS	For each element with a linear location: its relation specified by associated net elements considering the given application direction
LINEAR_LOCATION_SEQUENCE	Mapping of linear location IDs to their specified sequence numbers
LINEAR_LOCATION_ KEEPS_ORIENTATION	For each element with a linear location: boolean for each associated net element whether it keeps orientation
SPOT_LOCATION_REFERS_TO	Mapping of spot location IDs to their parent IDs
SPOT_LOCATIONS	For each element with spot locations: set of their spot locations
ALL_INTRINSIC_COORDINATES	Set of all locations induced by intrinsic coordinates of elements
NET_RELATION_ SUBSEQUENT_LOCATIONS	Topology Relation: connects all intrinsic coordinates from above set with the given net relations to one relation
railML3_IS_NET_RELATION_ SUBSEQUENT_LOCATIONS_closure1	Precomputed transitive closure of the topology relation
Interlocking (RailML3_IL_):	
CONFLICTING_ROUTES	Set of conflicting route IDs per route ID
DERAILER_REFERS_TO	Mapping of derailerIL to their derailerIS IDs
DERAILER_HAS_TVD_SECTION	Mapping of derailerIS to set of its TVD sections
DERAILER_IS_KEY_LOCKED	Boolean, indicating whether derailerIS is key locked
DERAILER_PREFERRED_POSITION	Preferred position of derailerIL
DERAILER_RETURNS_ TO_PREFERRED_POSITION	Boolean, indicating whether derailerIL returns to preferred position
DERAILER_RELATED_ MOVABLE_ELEMENT	Related movable element (crossing/derailer/switch) per derailerIS
DERAILER_MAX_THROW_TIME	maxThrowTime of each derailerIS in ms
DERAILER_TYPICAL_THROW_TIME	typicalThrowTime of each derailerIS in ms
MOVABLE_CROSSING_REFERS_TO	Mapping of movableCrossing to their crossing IDs
MOVABLE_CROSSING_IDS	Subset of crossing IDs that are movable
MOVABLE_CROSSING_BRANCHES	Function providing the branch relations per position of a movable crossing
MOVABLE_CROSSING_ HAS_TVD_SECTION	Mapping of movable crossing to set of its TVD sections

MOVABLE_CROSSING_IS_KEY_LOCKED	Boolean, indicating whether movable `crossing` is key locked
MOVABLE_CROSSING_PREFERRED_POSITION	Preferred position of `movableCrossing`
MOVABLE_CROSSING_RETURNS_TO_PREFERRED_POSITION	Boolean, indicating whether `movableCrossing` returns to preferred position
MOVABLE_CROSSING_RELATED_MOVABLE_ELEMENT	Related movable element (crossing/derailer/switch) per movable `crossing`
MOVABLE_CROSSING_MAX_THROW_TIME	`maxThrowTime` of each movable `crossing` in ms
MOVABLE_CROSSING_TYPICAL_THROW_TIME	`typicalThrowTime` of each movable `crossing` in ms
OVERLAP_LENGTH	Length of an overlap if specified
OVERLAP_VALIDITY_TIME	Validity time of an overlap
OVERLAP_NXT	Computed path of an overlap
OVERLAP_MUST_SWITCH_POSITIONS	Switch position of an overlap that *must* be proven
OVERLAP_CROSSING_POSITIONS	Required (derived) positions of crossings required for the overlap path
OVERLAP_DERAILERS	Passable derailers required (derived) for the overlap path
OVERLAP_SWITCH_POSITIONS	Required (derived) positions of switches required for the overlap path
OVERLAP_RELEASE_TIMERS	Release timer of an overlap if specified
OVERLAP_RELEASE_TRIGGER_SECTION	Release trigger sections (TVD sections) of an overlap if specified
ROUTE_RELEASE_GROUPS_AHEAD_TVD_SECTIONS	set of TVD sections of a route release group ahead
ROUTE_RELEASE_GROUP_AHEAD_NXT	Path of the route release groups ahead per route
ROUTE_RELEASE_GROUPS_AHEAD_TYPICAL_DELAYS	Typical delay of a route release group ahead
ROUTE_RELEASE_GROUPS_REAR_TVD_SECTIONS	set of TVD sections of a route release group rear
ROUTE_RELEASE_GROUP_REAR_NXT	Path of the route release groups rear per route
ROUTE_RELEASE_GROUPS_REAR_TYPICAL_DELAYS	Typical delay of a route release group rear
ROUTE_RELATION_MUST_DERAILER_POSITIONS	Derailer positions of a route relation that *must* be proven
ROUTE_RELATION_MUST_SECTION_STATES	TVD section states of a route relation that *must* be proven
ROUTE_RELATION_MUST_SWITCH_POSITIONS	Switch positions of a route relation that *must* be proven

`ROUTE_FORCED_ SWITCH_POSITIONS`	Switch positions specified for a route (e.g. facing/trailing switches)
`ROUTE_ENTRY`	Entry of a route as pair of the involved element and its location
`ROUTE_EXIT`	Exit of a route as pair of the involved element and its location
`ROUTE_NXT`	Computed path of a route
`ROUTE_CROSSING_POSITIONS`	Required (derived) positions of crossings required for the route path
`ROUTE_DERAILERS`	Passable derailers required (derived) for the route path
`ROUTE_SWITCH_POSITIONS`	Required (derived) positions of switches required for the route path
`ROUTE_RELEASE_GROUPS`	Set of route release groups of a route if specified
`ROUTE_OVERLAPS`	Set of overlaps of a route if specified
`ROUTE_ADDITIONAL_RELATIONS`	Set of additional route relations of a route if specified
`ROUTE_LOCKS_AUTOMATICALLY`	Boolean, indicating whether a route locks automatically
`ROUTE_TVD_SECTIONS`	Specified and derived TVD sections of a route
`ROUTE_ACTIVATION_SECTION_ TVD_SECTIONS`	Set of specified TVD sections of an activation section
`ROUTE_ACTIVATION_SECTIONS`	Set of specified and derived activation sections of a route
`SIGNAL_REFERS_TO`	Mapping of `signalIL` to their `signalIS` IDs
`SIGNAL_CONTROLLED`	Subset of `signalIS` IDs that have an interlocking object and are considered as controllable
`SIGNAL_END_OF_ CONTROL_SECTION`	Set of elements that are an end of the control section of a controllable signal
`SIGNAL_CONTROL_SECTIONS`	Paths of the control sections of a controllable signal per end element
`SIGNAL_CONTROL_TVD_SECTIONS`	TVD sections of the control sections of a controllable signal per end element
`SWITCH_REFERS_TO`	Mapping of `switchIL` to their `switchIS` IDs
`SWITCH_BRANCHES`	Function providing the branch relations per position of a switch
`SWITCH_POSITION_ RESTRICTIONS`	Position restrictions of a `switchIL` as record for derailer and switches per position
`SWITCH_IS_KEY_LOCKED`	Boolean, indicating whether `switchIS` is key locked
`SWITCH_PREFERRED_POSITION`	Preferred position of `switchIL`
`SWITCH_RETURNS_ TO_PREFERRED_POSITION`	Boolean, indicating whether `switchIL` returns to preferred position
`SWITCH_HAS_TVD_SECTION`	Mapping of `switchIS` to set of its TVD sections
`SWITCH_HAS_FOOLING_ TRAIN_DETECTORS`	Set of fooling train detectors of a `switchIS` if specified

`SWITCH_RELATED_MOVABLE_ELEMENT`	Related movable element (crossing/derailer/switch) per `switchIS`
`SWITCH_MAX_THROW_TIME`	`maxThrowTime` of each `switchIS` in ms
`SWITCH_TYPICAL_THROW_TIME`	`typicalThrowTime` of each `switchIS` in ms
`TVD_SECTIONS`	Computed paths of a TVD section as one relation
`TVD_SECTION_BERTHING_TRACK`	Boolean, indicating whether a TVD section is a berthing track
`TVD_SECTION_DEMARCATING_ELEMENTS`	Set of demarcating elements of a TVD section
`ASPECT_RELATION_APPLIES_TO_ROUTES`	Set of routes to which an aspect relation applies
`ASPECT_RELATION_SIGNAL_ASPECTS`	Specified aspect relations as a record of master, slave and distant per ID
`ASPECT_RELATION_SIGNAL_ASPECTS_INFERRED_FROM_ROUTE`	Set of inferred aspect relations per route
`SIGNAL_NOT_CONTROLLED_BY_SIGNALPLAN`	Set of signals that are not controlled by any signal plan
`SPECIFIC_INFRASTRUCTURE_MANAGER_GENERIC_ASPECT_OF_ID`	Mapping of an signal aspect ID to its generic aspect
Common (`RailML3_CO_`):	
`NAMES`	Provides the specified name of an element for a given ID together with a language (e.g. “en” for English)
Visualization (`RailML3_VIS_`):	
`NET_ELEMENT_COORDINATES`	Provides a function of subsequent virtual nodes for linear elements projections forming one net element (mainly for `D4R_customGraph`)

Table 7: List of Chosen Default Values

Type	Default Values
`aspectRelation`	`endSectionTime`: 1000
`derailerIL`	`maxThrowTime`: 2000; `typicalThrowTime`: 1000
`level`	`descriptionLevel`: *Micro*
`linearLocation`	`intrinsicCoordinateBegin`: *derived value,* 0.0 *if not possible*; `intrinsicCoordinateEnd`: *derived value,* 1.0 *if not possible*
`movableCrossing`	`maxThrowTime`: 2000; `typicalThrowTime`: 1000
`netRelation`	`navigability`: *both*; `positionOnA`: 1; `positionOnB`: 0
`overlap`	`mustOrShould@requiresSwitchInPosition`: *must*; `overlapValidityTime`: 1000; `proving@requiresSwitchInPosition`: *oneOff*; `timerValue@overlapReleaseTimer`: 1000
`route`	`delayForLock@routeActivationSection`: 1; `automaticReleaseDelay@routeActivationSection`: 1
`routeRelation`	`mustOrShould@requiredDerailerPosition`: *must*; `proving@requiredDerailerPosition`: *oneOff*; `mustOrShould@requiredDetectorState`: *must*; `proving@requiredDetectorState`: *oneOff*; `mustOrShould@requiredSectionState`: *must*; `proving@requiredSectionState`: *oneOff*; `mustOrShould@requiredSwitchPosition`: *must*; `proving@requiredSwitchPosition`: *oneOff*
`routeRelease-GroupAhead/-Rear`	`typicalDelay`: 1000
`switchIL`	`maxThrowTime`: 2000; `typicalThrowTime`: 1000
`spotLocation`	`applicationDirection`: *both*; `intrinsicCoordinate`: *derived value,* 0.0 *if not possible*

Table 8: List of Validated Semantic Properties – if violated: E: Error, W: Warning

Type		Validated Properties
`border`	$\boxed{\text{E}}$	If `openEnd=true`, it is actually located at an open track end
	-	If `openEnd=true`, there is no `bufferStop` at the same location
`bufferStop`	$\boxed{\text{E}}$	Is located at an open track end
`derailerIL`	$\boxed{\text{E}}$	If `returnsToPreferredPosition=true`, such a position is defined
	$\boxed{\text{W}}$	`hasTvdSections` should be part of the derailer's track parts
`level`	$\boxed{\text{E}}$	Each `netResource` is assigned to only one level type
	-	`elementA` and `elementB` of a `netRelation` are on the same level as the relation itself
	$\boxed{\text{W}}$	Micro level is not empty (animation not, limited validation possible)
`linearLocation`	$\boxed{\text{E}}$	For `track`, `platform`: location is connected in topology (also via navigability `None`)
	-	Sequence has no gaps and matches the order induced by the topology
	$\boxed{\text{W}}$	Referenced `netElements` have a length – needed for computation of intrinsic coordinates if they are not specified – otherwise they may be imprecise
`movableCrossing`	$\boxed{\text{E}}$	If `returnsToPreferredPosition=true`, such a position is defined
	-	Branches match branches of the referenced `crossing`
	$\boxed{\text{W}}$	`hasTvdSections` should be part of the crossing's track parts
`netElement`	$\boxed{\text{W}}$	Each `intrinsicCoordinate` is specified only once
`overlap`	$\boxed{\text{E}}$	There is a unique path between the `relatedTrackAssets` and `limitedBy`, taking into account the fixed switch positions
	-	`requiresSwitchInPosition` is unique for each `switchIL`
	-	Derived length is greater or equal than the specified minimal length
	$\boxed{\text{W}}$	`hasTvdSections` should intersect the overlap's path

`route`	$\boxed{\text{E}}$ Route entry and exit are connected by the topology - Route entry and exit are not referencing the same element - Forced switch positions establish a unique route path - Forced switch positions match to the topology between entry and exit - Forced switch position is unique for each `switchIL` - Derailer positions of additional `routeRelation`s match to the route path - Switch positions of additional `routeRelation`s match to the route path - Selected properties of Abrial's model: • Entry and exit are route blocks • Route path without entry and exit is a total bijection $\boxed{\text{W}}$ Abrial: Routes are not overlapping - `hasTvdSection`s should intersect the route's path
`routeRelation`	$\boxed{\text{E}}$ `requiredDerailerInPosition` is unique for each `derailerIL` - `requiredSectionInState` is unique for each `tvdSection` - `requiredSwitchPosition` is unique for each `switchIL`
`routeRelease-GroupAhead/-Rear`	$\boxed{\text{E}}$ Specified `tvdSection`s induce a valid path $\boxed{\text{W}}$ `hasTvdSection`s should intersect the group's path
`signalBox`	$\boxed{\text{W}}$ For each `aspectRelation` with a certain `master` aspect, there exists at least one `aspectRelation` having a `slave` showing the same aspect (i.e. the relations are "connectable") - If a `aspectRelation` contains a signal showing `aspect_closed`: `expectingSpeed=0`
`signalIL`	$\boxed{\text{E}}$ Valid path of controlled section to next `bufferStop`, open end, or `signalIL` could be inferred
`spotLocation`	$\boxed{\text{E}}$ All elements of type `balise`, `baliseGroup`, `border`, `bufferStop`, `crossing`, `derailerIS`, `levelCrossingIS`, `trainDetectionElement`, `trainProtectionElement`, `signalIS`, and `switchIS` are expected to have exactly one `spotLocation` - No `derailerIS`, `crossing`, or `switchIS` have the same location

`switchIS`	$\boxed{\text{E}}$ Left and right branch are connected - Turning branch end is connected to a straight branch end
`switchIL`	$\boxed{\text{E}}$ If `returnsToPreferredPosition=true`, such a position is defined - Interlocking branches match infrastructure branches - If related `switchIS` has `type=singleSwitchCrossing`: `hasPositionRestriction` must be specified for correct definition of the not switchable branch - If switch has `relatedMovableElement` of type `derailerIL`: `hasPositionRestriction` must be specified for that derailer $\boxed{\text{W}}$ `hasTvdSections` should intersect at least one switch branch - If `refersTo` references `crossing`: element is ignored
`track`	$\boxed{\text{E}}$ Track begin/end inferred from the topology matches the explicitly specified one - If track begins/ends with a `netElement`: begin/end is not ambiguous (i.e. neither crossing nor switch) - Track begin and end element are uniquely connected by the topology $\boxed{\text{W}}$ Specified track length matches sum of the lengths of its `netElements`
`tvdSection`	$\boxed{\text{E}}$ Each section has at least one demarcating train detector or at least two demarcating elements in case of mixed types or bufferstops only - Given the demarcating elements, a path can be established between them $\boxed{\text{W}}$ In case of missing demarcating element: open end was derived from topology - All demarcating elements are located at an end of the section - All `netElements` are part of a `tvdSection` - `tvdSections` do not overlap

D Variables and Operations for Animation

Table 9: List of Variables for Animation

Name	Description
`IS_next`	relation containing the currently passable track
`IS_crossing_states`	current states of movable crossings
`IS_crossingsInMovement`	movable crossings that are currently in movement
`IS_crossing_keyLocked`	key locking state of movable crossings
`IS_derailer_states`	current states of derailers
`IS_derailersInMovement`	derailers that are currently in movement
`IS_derailer_keyLocked`	key locking state of derailers
`IS_switch_states`	current states of switches
`IS_switchesInMovement`	switches that are currently in movement
`IS_switch_keyLocked`	key locking state of switches
`RS_requestingArrivalTrains`	arriving trains together with the location of the corresponding open end
`RS_arrivedTrains`	trains that have been arrived and are currently controlled by the interlocking
`RS_trainOccupiedLocations`	set of occupied locations without directions per train
`RS_trainFront`	current front location of each train
`RS_trainBack`	current back location of each train
`IL_occupiedTvdSections`	IDs of occupied TVD sections together with the occupying train
`IL_routes_in_res`	IDs of routes that are currently in reservation together with the reserving train
`IL_res_routes`	IDs of reserved routes together with the reserving train
`IL_res_route_blocks`	blocks (locations) that are currently reserved by routes together with the reserving route
`IL_res_blocks`	set of blocks (locations) that are currently reserved by routes
`IL_released_partialRoutes`	IDs of released routeReleaseGroups per route
`IL_res_overlaps`	IDs of reserved overlaps together with the reserving train
`IL_overlaps_in_release`	IDs of reserved overlaps that are currently in release process
`IL_signal_states`	current states of signals (i.e. their aspects)
`IL_noted_signal_states`	noted states for change of signals that are applied when the corresponding signal plan is activated
`IL_crossing_locked_routes`	IDs of movable crossings locked by a route together with the route ID
`IL_derailer_locked_routes`	IDs of derailers locked by a route together with the route ID
`IL_switch_locked_routes`	IDs of switches locked by a route together with the route ID
`IL_signal_locked`	IDs of signals locked by a signal plan
`IL_signalplan_in_activation`	set of signal plans in activation

Table 10: List of Operations for Animation

Name	Parameters/Description
IS_startChangeDerailer	(DerailerId, DestState) start movement of derailer with ID DerailerId to DestState
IS_endChangeDerailer	(DerailerId) end movement of derailer DerailerId
IS_startChangeCrossing	(CrossingId, CurrState, DestState) start movement of movable crossing with ID CrossingId from CurrState to DestState
IS_endChangeCrossing	(CrossingId) end movement of crossing CrossingId
IS_startChangeSwitch	(SwitchId, CurrState, DestState) start movement of not coupled ordinary switch or switch crossing with ID SwitchId from CurrState to DestState
IS_endChangeSwitch	(SwitchId) end movement of any switch with SwitchId
IS_startChangeCoupledSwitches	(SwitchId1, CurrState1, DestState1, SwitchId2, CurrState2, DestState2) start simultaneous movement of coupled ordinary switches with IDs SwitchId1/2 from CurrState1/2 to DestState1/2; end is performed individually
IL_startRouteReservation	(Route, Train) starts reservation of Route for the requesting Train if all preconditions are fulfilled (route and related overlaps are free, train is allowed to request the route)
IL_endRouteReservation	(Route, Train) end reservation of Route for the requesting Train if all preconditions are fulfilled (route relations, switch positions of routes and overlaps, possible signal plan is in activation)
IL_partialRouteReleaseAhead	(Route, routeReleaseGroup) release routeReleaseGroup ahead that is part of reserved Route if train has not yet arrived at the release group
IL_completeRouteReleaseAhead	(Route) completely release Route ahead if train has not yet passed the route, is on a berthing track, and all release groups ahead are already released
IL_partialRouteReleaseRear	(Route, routeReleaseGroup) release routeReleaseGroup rear that is part of reserved Route if train has completely passed the release group
IL_completeRouteReleaseRear	(Route) completely release Route ahead if train has passed the entire route and all release groups rear are already released
IL_startOverlapRelease	(OverlapId) start release of OverlapId if a train occupies one of its release trigger sections
IL_endOverlapRelease	(OverlapId) end release of OverlapId if it is in release
IL_startActivateSignalplan	(Signalplan) start activation of Signalplan that does not conflict with another plan in activation and if it applies to a route, the route must be in reservation

`IL_endActivateSignalplan`	(`Signalplan`) end activation of `Signalplan` after noting all signal changes
`IL_noteChangeSignalState`	(`SignalId`) when a signal plan is in activation, signals can be noted for change after application of the signal plan if the signal section is passable and not occupied
`IL_changeSignalState`	(`SignalId`) the state of individual signals not controlled by signal plans can be directly changed if the signal section is passable and not occupied
`RS_trainArrivalRequest`	(`OpenendId`, `Train`, `Position`) a `Train` can request arrival at an `OpenendId` with `Position` if the corresponding TVD section is vacant
`IL_trainAcceptArrival`	(`Train`) a `Train` that has requested for arrival can be accepted and enter the interlocking area
`IL_trainDeclineArrival`	(`Train`) a `Train` that has requested for arrival can be declined without affecting the interlocking area
`RS_trainLeave`	(`Train`) a `Train` that has fully approached an open end can leave the interlocking area, which will also release all its reserved routes
`RS_trainMoveFront`	(`currFront`, `newFront`, `Train`) the front of a `Train` can move one step in the topology relation from `currFront` to `newFront`
`RS_trainMoveBack`	(`currBack`, `newBack`, `Train`) the back of a `Train` can move one step in the topology relation from `currBack` to `newBack`
`RS_trainChangeDirection`	(`Train`) a `Train` that is on a berthing track and has released all its routes can change direction

E Internal Representation of B-Rules DSL

Listing 29: Example of an arbitrary Rule in B-Rules DSL

```
RULE rule2
DEPENDS_ON_RULE rule1
DEPENDS_ON_COMPUTATION comp1
BODY
    RULE_FORALL i
        WHERE i : 1..10
    EXPECT i > 5
    COUNTEREXAMPLE STRING_FORMAT("~w <= 5", i)
    END
END
```

Listing 30: Internal Representation of the Rule in Listing 29 in Classical B

```
`$RESULT`,`$COUNTEREXAMPLES` <-- rule2 =
    SELECT
        rule2 = "NOT_CHECKED"
        & comp1 = "EXECUTED"
        & rule1 = "SUCCESS"
    THEN
        VAR `$ResultTuple`,`$ResultStrings` IN
            `$ResultTuple` := FORCE({i|i : 1 .. 10 & not(i > 5)});
            `$ResultStrings` := FORCE({`$String`|`$String` : STRING
                & #i.(i : `$ResultTuple` & `$String` =
               FORMAT_TO_STRING("~w <= 5",[TO_STRING(i)]))});
            rule2_Counterexamples := rule2_Counterexamples
               \/ {1} * `$ResultStrings`;
            IF `$ResultTuple` /= {} THEN
                rule2,`$RESULT`,`$COUNTEREXAMPLES` := "FAIL",
                    "FAIL",rule2_Counterexamples
            END
        END;
        IF rule2 /= "FAIL" THEN
            rule2,`$RESULT`,`$COUNTEREXAMPLES` := "SUCCESS",
               "SUCCESS",{}
        ELSE
            PRINT(rule2_Counterexamples)
        END
    END
```

List of Figures

List of Tables

List of Listings

References

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 2010.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[Ben+21] Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, and Michelle Werth. "ProB2-UI: A Java-Based User Interface for ProB". In: *Formal Methods for Industrial Critical Systems*. Ed. by Alberto Lluch Lafuente and Anastasia Mavridou. Cham: Springer International Publishing, 2021, pp. 193–201. ISBN: 978-3-030-85248-1.

[Bol20] Ya-Chun Bollig. "Geometrical and Topological Linking of Railway Systems". Master's thesis. Munich: Technical University of Munich, 2020.

[Bra23] Torben Brand. "ISO RailDax timeline and railML usage in Norway". 43rd railML Conference, Berlin. May 2023.

[But+20] Michael Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia, and Laurent Voisin. "The First Twenty-Five Years of Industrial Use of the B-Method". In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2020, pp. 189–209.

[Cap+17] Quentin Cappart, Christophe Limbrée, Pierre Schaus, Jean Quilbeuf, Louis-Marie Traonouez, and Axel Legay. "Verification of Interlocking Systems Using Statistical Model Checking". In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering*. IEEE. 2017, pp. 61–68.

[CNC17] T Ciszewski, W Nowakowski, and M Chrzan. "RailTopoModel and RailML – data exchange standards in railway sector". In: *Archives of Transport System Telematics* 10 (2017).

[FB22] Alessio Ferrari and Maurice H. Ter Beek. "Formal Methods in Railways: a Systematic Mapping Study". In: *ACM Computing Surveys* 55.4 (2022), pp. 1–37.

[GBO18] Tim Gonschorek, Ludwig Bedau, and Frank Ortmeier. "Bringing formal methods on the rail: On automatic verifying railroad interlockings from railML models". In: *Safety and Reliability – Safe Societies in a Changing World: Proceedings of ESREL 2018, June 17-21, 2018, Trondheim, Norway*. Ed. by Stein Haugen, Anne Barros, Coen Gulijk, Trond Kongsvik, and Jan Erik Vinnem. CRC Press, 2018, pp. 741–748.

[Gra23] *Graphviz*. URL: https://graphviz.org/ (accessed on 22/6/2023).

[Gru+23] Jan Gruteser, David Geleßus, Michael Leuschel, Jan Roßbach, and Fabian Vu. “A Formal Model of Train Control with AI-Based Obstacle Detection”. In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - 5th International Conference, RSSRail 2023, Berlin, Germany, 10-12 October, 2023, Proceedings*. Ed. by Birgit Milius, Simon Collart Dutilleul, and Thierry Lecomte. Springer, 2023, pp. 128–145.

[Hei18] Christoph Heinzen. “A user-interface Plugin for the Rule Validation Language in ProB”. Master’s thesis. Düsseldorf: Heinrich-Heine-University, 2018.

[Hlu17] Adam Hlubuček. “RailTopoModel and RailML 3 in Overall Context”. In: *Acta Polytechnica CTU Proceedings* 11 (2017), pp. 16–21.

[HSL16] Dominik Hansen, David Schneider, and Michael Leuschel. “Using B and ProB for Data Validation Projects”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016*. Springer, 2016, pp. 167–182.

[Jer23] *Infrastructure model – a digital description of the railway*. URL: https://www.jernbanedirektoratet.no/en/about-us/infrastructure-model--a-digital-description-of-the-railway/ (accessed on 26/6/2023).

[Kol+23] Vasco Paul Kolmorgen, Christian Rahmig, Jörg von Lingen, and Milan Wölke. “The Federal Ministry of Transport’s digitalisation strategy for regional railways”. In: *Signalling & Datacommunication* 1+2 (2023), pp. 6–13. URL: https://elib.dlr.de/196688/1/06_13_Kolmorgen_etal.pdf.

[Kol23] Vasco Paul Kolmorgen. “Governance & News”. 43rd railML Conference, Berlin. May 2023.

[Kör+21] Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. “Integrating formal specifications into applications: the ProB Java API”. In: *Formal Methods in System Design* 58.1-2 (2021), pp. 160–187.

[LB03] Michael Leuschel and Michael Butler. “ProB: A Model Checker for B”. In: *FME 2003: Formal Methods*. Vol. 2805. LNCS. Berlin, Heidelberg: Springer, Sept. 2003, pp. 855–874.

[Lec+17] Thierry Lecomte, David Déharbe, Étienne Prun, and Erwan Mottin. “Applying a Formal Method in Industry: a 25-Year Trajectory”. In: *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29—December 1, 2017, Proceedings 20*. Springer. 2017, pp. 70–87.

[Leu23] Michael Leuschel. “ProB: Harnessing the Power of Prolog to Bring Formal Models and Mathematics to Life”. In: *Prolog: The Next 50 Years*. Springer, 2023, pp. 239–247.

[LJS16] Bjørnar Luteberget, Christian Johansen, and Martin Steffen. “Rule-based Consistency Checking of Railway Infrastructure Designs”. In: *Integrated Formal Methods: 12th International Conference, IFM 2016*. Ed. by Erika Ábrahám and Marieke Huisman. Springer. 2016, pp. 491–507.

[LM16] Thierry Lecomte and Erwan Mottin. “Formal Data Validation in the Railways”. In: *Proceedings of the Twenty-fourth Safety-Critical Systems Symposium, Brighton, UK*. Ed. by Mike Parsons and Tom Anderson. Feb. 2016, pp. 355–364.

[Mar+22] João Martins, José M. Fonseca, Rafael Costa, José C. Campos, Alcino Cunha, Nuno Macedo, and José N. Oliveira. “Verification of Railway Network Models with EVEREST”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 2022, pp. 345–355.

[Men+23] Martín Nicolás Menéndez, Santiago Germino, Luis David Díaz-Charris, and Ariel Lutenberg. “Automatic Railway Signaling Generation for Railways Systems Described on Railway Markup Language (railML)”. In: *IEEE Transactions on Intelligent Transportation Systems* (2023), pp. 1–11. DOI: 10.1109/TITS.2023.3317256.

[Nas+04] Andrew Nash, Daniel Huerlimann, Jörg Schütte, and Vasco Paul Krauss. “RailML – a standard data interface for railroad applications”. In: *Computers in Railways IX*. Vol. 74. WIT Transactions on The Built Environment. WIT Press, 2004, pp. 233–240.

[PK21] Cheng Peng and Wang Keming. “Applying B and ProB to a Real-world Data Validation Project”. In: *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. IEEE. 2021, pp. 521–524.

[Rai23a] *Home – railML.org*. URL: https://www.railml.org/ (accessed on 31/8/2023).

[Rai23b] *railML 3 Wiki*. URL: https://wiki3.railml.org/ (accessed on 31/8/2023).

[Rai23c] *Home – RailTopoModel*. URL: https://www.railtopomodel.org/ (accessed on 31/8/2023).

[Rut23] Kristin Rutenkolk. “Extending Modelchecking with ProB to Floating-Point Numbers and Hybrid Systems”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2023, pp. 366–370.

[St-23] Richard St-Denis. “A comparison of three solver-aided programming languages: αRby, ProB, and Rosette”. In: *Journal of Computer Languages* 77 (2023). DOI: https://doi.org/10.1016/j.cola.2023.101238.

[VHL22] Fabian Vu, Christopher Happe, and Michael Leuschel. “Generating Domain-Specific Interactive Validation Documents”. In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2022, pp. 32–49.

[VL23] Fabian Vu and Michael Leuschel. “Validation of Formal Models by Interactive Simulation”. In: *Rigorous State-Based Methods: 9th International Conference, ABZ 2023, Nancy, France, Proceedings*. Ed. by Uwe Glässer, Jose Creissac Campos, Dominique Méry, and Philippe Palanque. Vol. 14010. LNCS. Springer. 2023, pp. 59–69.

[VLM21] Fabian Vu, Michael Leuschel, and Atif Mashkoor. “Validation of Formal Models by Timed Probabilistic Simulation”. In: *Rigorous State-Based Methods: 8th International Conference, ABZ 2021, Ulm, Germany, Proceedings.* Ed. by Alexander Raschke and Dominique Méry. Vol. 12709. LNCS. 2021, pp. 81–96.

[WJ17] Susanne Wunsch and Birgit Jaekel. “Modellprinzipien des RailTopoModel”. In: *Der Eisenbahningenieur* 03 (2017), pp. 18–23.

[WL20] Michelle Werth and Michael Leuschel. “VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics”. In: *Rigorous State-Based Methods: 7th International Conference, ABZ 2020, Ulm, Germany, Proceedings.* Ed. by Alexander Raschke, Dominique Méry, and Frank Houdek. Vol. 12071. LNCS. 2020, pp. 260–265. ISBN: 978-3-030-48077-6.