Technical Report

- **Project**: DigiLightRail
- **WP**: WWW
- **Deliverable**: T1.4
- **Producer**: HASLab / INESC TEC
- **Title**: Verification of Railway Network Models
- **Summary**: *This report (.....)*

Overview

This technical report is part of work package WP WWW of the contract CCC signed between XXX and YYY concerning project PPP. The main aim of this report is to present a brief overview of formal modeling and specification techniques for software design and then show how these can be effectively applied to the verification of railway network models and, in particular, to railML designs, which is the option relevant to the project.

The CENELEC EN 50128 standard[1] imposes conditions for applying formal verification tools to safety critical software in the railway domain. Formal methods (FM) are highly recommended for safety-critical projects to reach the highest safety integrity levels (SIL) of SIL3 and SIL4. CENELEC EN 50128 classifies tools into three different types, from T1 to T3, depending on whether they can introduce faults into the safety critical software[2]. The standard also requires that, in safety-critical applications: (1) the choice of tools within the particular level at target (T1 to T3) is justified; (2) potential failures are identified, as well as mitigation procedures to handle them; (3) the chosen tool has a specification or handbook; (4) only justified versions of the tools are used and (5) the previous requirement carries over when switching versions of the tool.

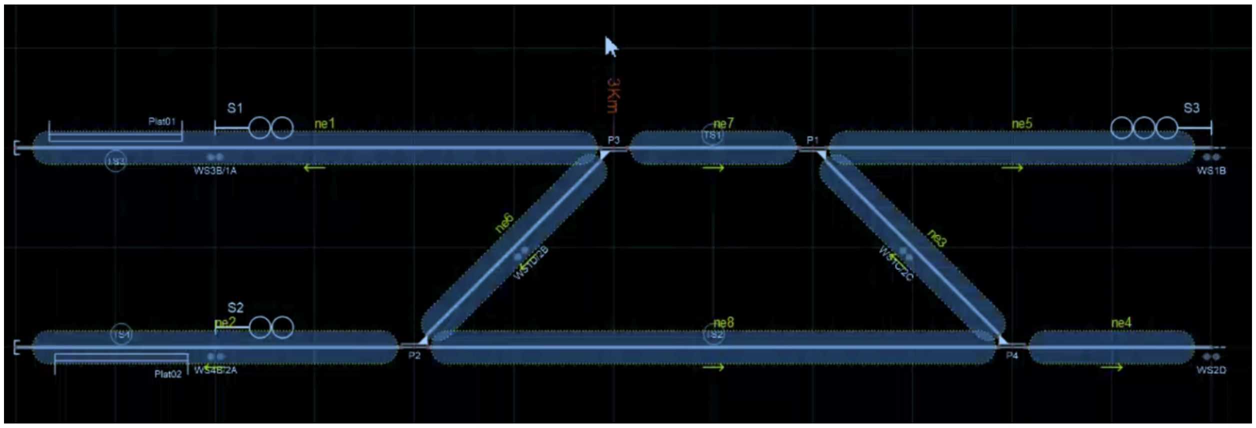
To the best of our knowledge, FMERAIL (1998-1999) was the first European project applying formal methods in the railways domain, coordinated by Instituttet for Anvendt Datateknik, Denmark (FP4-ESPRIT 4). Since then, many research institutions and companies have sought to use formal techniques in the design of railways software, including work by us[3]. However, a

[1] EN 50128:2011 “Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems”. CENELEC CLC/TC 9X standard, 2011-06.
[2] FM-based verification tools are usually of type T2 (verification may fail to identify faults but cannot introduce them themselves).
[3] Alcino Cunha, Nuno Macedo: Validating the Hybrid ERTMS/ETCS Level 3 concept with Electrum. Int. J. Softw. Tools Technol. Transf. 22(3): 281-296 (2020)

relatively recent survey[4] concluded that no universally accepted formal framework has emerged, and railway companies wishing to adopt formal methods still have little guidance in the selection of the most appropriate methods and tools. 4SECURAIL (2019-) is an on-going EU project in this field coordinated by Ardanuy Ingeniería, S.A. and funded under the Shift2Rail initiative. Former European projects in the domain include RobustRailS (2012-16) and ASTRail (2017-19), the latter also funded by Shift2Rail. A number of research institutions have joined the European Technical Working Group on Formal Methods in Railway Control (although the group does not seem active since 2017). The RSSRail conference series is the current forum for discussions in the field[5]. Current topics of interest include formal modelling, verification techniques and validation according to the standards[6].

This report will briefly survey how formal methods can be used to verify railML® network models. It starts by introducing railML, both from the infrastructure and interlocking perspective, the two relevant views of a railway network for the purpose of the DigiLightRail project. We then present a primer on formal specification and verification addressing the two most relevant logics for specifying safety properties: *First-Order Logic* for structural properties and *Temporal Logic* for behavioral properties. We also discuss verification techniques needed to check the validity of assertions specified on those logics. Finally we discuss the (scarce) existing work on verifying railML network models.

For illustration purposes throughout the report we will use a simple example of a railway station, whose layout is depicted below. This station comprises two parallel tracks, four switches to enable trains to be guided from one track to the other, three signals, and six train detection devices, which altogether define four train detection sections of interest for interlocking.



[4] Alessio Ferrari et al: Survey on Formal Methods and Tools in Railways - The ASTRail Approach, RSSRail 2019, 226-241. This survey was carried out as part of the ASTRail project.
[5] The next event in the series is scheduled for November 15th-16th 2021, in Paris.
[6] Luteberget, B., Johansen, C. Efficient verification of railway infrastructure designs against standard regulations. *Form Methods Syst Des* **52,** 1–32 (2018). https://doi.org/10.1007/s10703-017-0281-z

Railway network modeling with railML®

railML (Railway Markup Language) is an open, XML-based data exchange format for data interoperability of railway applications. As of version 3.1[7], railML includes four main subschemas:

- *Infrastructure*, for describing the railway network infrastructure;
- *Interlocking*, for describing railway signalling and interlocking systems;
- *Rollingstock*, for describing rail vehicles and formations;
- *Timetable and Rostering*, for describing railway timetables.

For the purpose of DigiLightRail, we are mainly interested in the Infrastructure and Interlocking subschemas, which will be briefly presented below. A more detailed presentation of these subschemas can be found in the respective railML 3.1 Tutorials[8][9]. Some railway design tools, such as RaIL-AiD[10], already export models of the network in the railML format. (The diagram of our train station example in the previous section was produced by this tool.)

The model of a railway infrastructure, described in railML inside a single tag `<infrastructure>`, is divided into two main parts: topology and functional infrastructure. The topology, described inside tag `<topology>`, describes the structure of a railway track network. The railML encoding of the topology follows the *RailTopoModel*® (RTM) standard topological model[11]. A railML model can describe the topology of several networks inside tag `<networks>`, at different levels of abstraction, ranging from a *microscopic* view that describes how track segments are connected at switches or crossings to a *mesoscopic* or *macroscopic* level, where the focus is on railway lines and their interconnection at railway stations or other operational points. (The DigiLightRail project is mainly concerned with the microscopic level.) In a railML topological model, track segments are defined using tag `<netElement>` and a connection between two track segments is defined by a `<netRelation>`. A snippet of the railML model of our example train station is listed below. As one can see in the depiction above, the track segment identified as `ne1` is connected to two other track segments, identified as `ne6` and `ne7`, at switch `P3`. As such, in the XML snippet, we can see that the `<netElement>` describing this track segment has two children `<relation>` elements that point to the `<netRelation>` elements that will describe the two connections. As an example, the XML snippet also includes one `<netRelation>` element that describes one of the connections, namely the connection between track segments `ne1` and `ne7`, identified inside children `<elementA>` and `<elementB>`, respectively. The `<netRelation>` element also indicates using attributes which endpoints of the two track segments are connected (with `0` denoting one endpoint and `1` the other). A `<netElement>` can also have an optional attribute with it's `length`. In this example, `ne1` has a length of 30 meters. Having described all track segments and their connections, a network can

[7] https://wiki3.railml.org
[8] Christian Rahming: *railML® 3.1 Tutorial - Simple Example Step-by-Step. Part 1: Infrastructure*. 2018.
[9] J. von Lingen: *railML® 3.1 Tutorial - Simple Example Step-by-Step. Part 2: Interlocking*. 2018.
[10] https://www.rail-aid.com
[11] http://www.railtopomodel.org/

be described inside tag `<network>` by just listing all the network resources that constitute it (a resource being either a `<netElement>` or a `<netRelation>`) and specifying its level of abstraction inside tag `<level>`. The snippet includes a single network described at the microscopic level.

The functional infrastructure, described inside tag `<functionalInfrastructure>`, describes all functional infrastructure elements that belong to a railway network. Among these we have physical elements such as switches, signals, train detection elements, or buffer stops, but also "virtual" elements such as border points to other zones of the network not being modelled, or maximum speed zones. Most elements specify the track segment in which they are located using tag `<spotLocation>` that includes a mandatory attribute pointing to the respective `<netElement>` and other optional attributes, namely attribute `pos` that can be used to specify the location (in meters) in respect to the endpoint 0 of the `<netElement>`. For illustration purposes, the railML snippet includes a `<signalIS>` element that describes signal `S1`[12], the signal located on track segment `ne1`, located 20 meters away from the start of the respective segment. It also includes a `<switchIS>` element describing switch `P3` located at the intersection of `ne1`, `ne7`, and `ne6`. The location of this switch is given in respect to the `<netElement>` facing it, namely `ne7`. Sub-elements `<leftBranch>` and `<rightBranch>` indicate to which `<netElement>`s the switch is connected, by including a reference to the respective `<netRelation>`s. The attributes `continueCourse` and `branchCourse` can then be used to discriminate which of the branches play each role.

```
<railML version="3.1">
   <infrastructure>
       <topology>
           <netElements>
               <netElement id="ne1" length="30">
                   <relation ref="nr_ne1ne7_swi7"/>
                   <relation ref="nr_ne1ne6_swi7"/>
               </netElement>
               ...
           </netElements>
           <netRelations>
               <netRelation id="nr_ne1ne7_swi7" positionOnA="0" positionOnB="0">
                   <elementA ref="ne1"/>
                   <elementB ref="ne7"/>
               </netRelation>
               ...
           </netRelations>
           <networks>
               <network id="nw01">
                   <level id="lv0" descriptionLevel="Micro">
                       <networkResource ref="ne1"/>
```

[12] Notice that in this case `S1` is not the unique identifier of the `<signalIS>`, defined in the attribute `id`, but a human readable identifier defined in a child element `<name>`.

```
                       <networkResource ref="nr_ne1ne7_swi7"/>
                       ...
                   </level>
               </network>
           </networks>
       </topology>
       <functionalInfrastructure>
            <signalsIS>
               <signalIS id="sig36">
                   <name name="S1" language="en"/>
                   <spotLocation id="sig36_loc-1" netElementRef="ne1" pos="20"/>
               </signalIS>
               ...
           </signalsIS>
           <switchesIS>
                <switchIS id="swi7" continueCourse="right" branchCourse="left">
                   <name name="P3" language="en"/>
                   <spotLocation id="swi7_sloc01" netElementRef="ne7" pos="0"/>
                   <leftBranch netRelationRef="nr_ne6ne7_swi7"/>
                   <rightBranch netRelationRef="nr_ne1ne7_swi7"/>
               </switchIS>
               ...
            </switchesIS>
           ...
       </functionalInfrastructure>
   </infrastructure>
   ...
</railML>
```

The model of the interlocking is described in railML inside a single tag `<interlocking>`. The subschema for the interlocking is still under development, as clearly mentioned in the railML wiki[13], but we will still present some of its key concepts. The network elements relevant for the interlocking are described inside tag `<assetsForIL>`. For some of the elements that were already described in the infrastructure some more detail is added in this sub-schema. That is the case, in particular, of signals and switches. For example, for signals some additional information can be added that is only relevant for the interlocking, such as whether the signal is virtual or not, or the various speeds allowed for the trains when approaching or passing the signal. While a signal in the infrastructure view is modeled inside tag `<signalIS>`, in the interlocking view the tag `<signalIL>` is used instead (likewise for switches). Below we present a snippet of the interlocking information for our example station. As we can see, some extra information is given concerning signal `S1` (whose `id` in the infrastructure is `sig36`), namely the `isVirtual` attribute is set to false and the `passingSpeed` is limited to 100. The link between the description of a signal in the interlocking and the respective description in the infrastructure is defined by the children tag <refersTo>.

[13] https://wiki3.railml.org

Another relevant element for interlocking is the one denoting *Train Vacancy Detection* (TVD) sections, sections of the track where it is possible to detect the presence of trains by means of train detection elements such as axle counters. In our running example we have 6 axle counters (depicted with two small circles next to each other) that partition the network in 4 TVD sections. Although not shown in the previous section, these train detection elements are also described in the infrastructure. A TVD is typically demarcated by train detection elements, but can also be demarcated by other elements such as buffer stops. For example, section `TS1` in the upper right part of layout is demarcated by 4 axle counters, while section `TS3` in the upper left part is demarcated by an axle counter and a buffer stop at the physical end of the line. A TVD section is described inside the tag `<tvdSection>`. Besides the usual mandatory unique identifier we have other mandatory attributes, namely delays for releasing a locked section. In the railML snippet we include the representation of section `TS3`, whose unique `id` is `ts3` and that has a `partialRouteReleaseDelay` of 4 seconds[14]. The children tag `<designator>` is used to store the section `entry` name in a particular `register`. The section demarcation elements, train detection elements or buffer stops, are also described inside the respective children tags.

```
<railML version="3.1">
   ...
   <interlocking>
       <assetsForIL id="stationX_assets">
           <signalsIL>
               <signalIL id="il_sig36" isVirtual="false" passingSpeed="100">
                   <refersTo ref="sig36"/>
               </signalIL>
               ...
           </signalsIL>
           <tvdSections>
               <tvdSection id="ts3" partialRouteReleaseDelay="PT4S" ...>
                   <designator register="stationX" entry="TS3"/>
                   <hasDemarcatingBufferstop ref="bus25"/>
                   <hasDemarcatingTraindetector ref="ac28"/>
               </tvdSection>
               ...
           </tvdSections>
           <routes>
               <route id="rt1">
                   <designator register="stationX" entry="Route S1-E21"/>
                   <facingSwitchInPosition id="rt1_swi6" inPosition="right">
                       <refersToSwitch ref="il_swi6"/>
                   </facingSwitchInPosition>
                   <hasTvdSection ref="ts1"/>
                   <hasTvdSection ref="ts2"/>
```

[14] This is the delay time after which the section may be released for use in a new route after a train has cleared it.

```
                    <routeEntry id="rt1_entry">
                        <refersTo ref="il_sig36"/>
                    </routeEntry>
                    <routeExit id="rt1_exit">
                        <refersTo ref="il_VirtSig50"/>
                    </routeExit>
                </route>
                ...
            </routes>
            ...
        </assetsForIL>
        ...
    </interlocking>
</railML>
```

The main goal of interlocking is to call and lock routes for trains to pass through railway stations or junctions. A route is a sequence of TVD sections between two signals. For a route to be locked it must be safe, that is, vacant of other trains and without any conflicting routes simultaneously locked. To lock a route, all its movable track elements (switches, level crossings, etc) must be locked until the train has cleared it. There are two main kinds of interlocking systems: geographical and tabular. In the former routes are found on-the-fly using some sort of shortest-path algorithm. The latter has a predefined set of routes, each listing the respective TVD section, and the required locking state for the movable elements. To support tabular interlocking, railML allows routes to be described with tag `<route>`. The above railML snippet shows the description of one of the possible routes in our example station, namely from signal `S1` to a virtual signal that is positioned at the border of the station in the bottom right corner. Likewise to TVD sections, routes can have an entry name in a register described inside children tag `<designator>`. Two mandatory children tags describe the `<routeEntry>` and `<routeExit>` signals. Using children tags `<hasTVDSection>` it is also possible to describe the sections the route traverses. In the case of the example route it traverses sections `TS1` and `TS2`. Finally we need to specify the required state of all relevant movable elements. For this particular route, switch `P1` (whose interlocking identifier is `il_swi6`) should be locked to the right branch, as described inside tag `<facingSwitchInPosition>`.

A primer on formal specification and verification

The verification of a railway design can address different facets, ranging from its topology and infrastructure to the interlocking. For each of these facets one may wish to specify and verify different safety rules, originating either from international and national safety regulations or from specific requirements of particular customers. Some examples of safety rules one may wish to specify and verify are:

1. Every signal is an entry point of a route. This rule imposes an infrastructure restriction on the railway network design, ensuring some sort of minimal coverage concerning the

possible routes. It is a static safety rule in the sense that, for a particular configuration of the network its validity is not affected by the passage of time.

2. Every signal is at least 20 meters away from the next switch. Similar to the previous one, this rule imposes a static restriction on the railway network design, ensuring enough distance after a signal to account for possible train overshoot (if it fails to stop on time before the signal). In this case we have a topological restriction referring to the distance (along the railway track) between two particular elements.
3. The entry signal of a route can only change its aspect from *closed* to *proceed* if all TVD sections along the route are *vacant*. This rule imposes an obvious safety constraint on the interlocking system, and unlike the previous rules it concerns the behaviour of the network, and its validity depends on the sequence of events that occur over time.

Static and behavioural rules require different logics to be specified. For static rules we need a logic that allows us to organize the elements of the domain of discourse in different *relations*, describing how elements are connected to each other, and that is expressive enough to allow *quantification* over this domain. *First-order logic*[15] is particularly well-suited for this purpose. For behavioural rules we need a logic that allows us to describe sequences of events or the evolution of the system state over time. For this we need some sort of *temporal logic*[16]. In the remainder of this section we will briefly describe these two logics, show how they can be used to formally specify the above rules, and present techniques that can be used to verify them.

First-order logic

The key concept in First-Order Logic (FOL) is that of *predicate*, a structure that somehow relates entities in the *domain of discourse* or *universe*. Hence, FOL is also known as *predicate logic*. Technically, a predicate is a set of tuples of entities of equal size (the predicate's *arity*), a mathematical structure also known as a *relation*. Predicates generalize the concept of *proposition* of *Propositional Logic* (PL)[17], which is an atomic sentence whose value is either true or false. To formalize a railML model in propositional logic we would need countless many propositions such as `S1_is_a_signal`, `S2_is_a_signal`, `S1_is_the_route_entry_of_RT1`, etc. With FOL we could have a universe $\mathcal{U}$ that contains all entities relevant to our model, in the case of railML all identifiers and numbers,

$$\mathcal{U} = \{S1,S2,S3,P1,P2,P3,P4,TS1,TS2,TS3,TS4,\ldots,0,1,2,3,4,5,\ldots,\infty\}$$

and then we could have different predicates to group these entities into categories or establish relations between them. For example we could have a unary predicate `signal` that groups together all entities that are identifiers of signals

```
signal = {(S1),(S2),(S3)}
```

15 https://en.wikipedia.org/wiki/First-order_logic
16 https://en.wikipedia.org/wiki/Temporal_logic
17 https://en.wikipedia.org/wiki/Propositional_calculus

and likewise for switches, TVD sections, and routes.

```
switch     = {(P1),(P2),(P3),(P4)}
tvdSection = {(TS1),(TS2),(TS3),(TS4)}
route      = {(RT1),(RT2),(RT3),(RT4),(RT5)}
```

Predicates of arity higher than 1 can then be used to relate different entities. For example, we could have a binary predicates `routeEntry` and `hasTvdSection`, that connect routes with the respective entry signals and TVD sections, respectively.

```
routeEntry    = {(RT1,S1),(RT2,S2),(RT3,S2),(RT4,S3),(RT5,S5)}
hasTvdSection = {(RT1,TS1),(RT1,TS2),(RT2,TS2),(RT3,TS2),(RT3,TS1),
                 (RT4,TS1),(RT4,TS3),(RT5,TS1),(RT5,TS2),(RT5,TS4)}
```

For numbers we could have the usual comparison binary predicates such as ≥.

Besides predicates we can also have *functions* in FOL, structures that associate a tuple of entities with a result entity. For example, for numbers we could have a binary function + that given a pair of numbers returns its sum. In the case of railML formalization we could have a binary function `distance` that given a pair of elements *x* and *y* of the network (signals or switches) returns the distance between them (or ∞ if *y* is not reachable from *x*).

```
distance(S1,P1) = 40
distance(S1,P2) = ∞
distance(S1,P3) = 20
distance(S1,P4) = 60
distance(S1,S1) = 0
distance(S1,S2) = ∞
distance(S1,S3) = ∞
distance(S2,P2) = 10
...
```

Notice that a function of arity *n* can also be seen as a predicate of arity *n+1* that relates the arguments with exactly one result, but unlike predicates functions can be used to directly specify complex *terms* denoting entities of the universe, where, for example, a function is applied to the result of another function. For example, a possible term in our example could be `distance(S1,P1)+distance(P1,P2)`.

All these predicates, with the exception of function `distance`, can directly be extracted from a particular railML model. For computing the `distance` between two elements we have to first extract a graph model of the network, more specifically a weighted directed graph, where nodes are either switches or signals (or other elements that can be positioned along tracks) and an edge with weight *d* exists between nodes *x* and *y* iff *x* and *y* are positioned in the same

`<netElement>` and *d* is the difference between their positions in that `<netElement>`. Notice that elements positioned at one endpoint of a `<netElement>` (namely switches) are also positioned at the respective endpoint of all `<netElement>`s connected to it. After obtaining this graph, the distance can be computed with any all shortest path algorithm such as Dijkstra[18] or, if we are interested in computing at once all distances, Floyd-Warshall[19].

Predicates and functions can be put together to build formulas using the operators described in the following table, which also presents their informal semantics.

Syntax	Name(s)	Informal semantics
$\top$	True	
$\bot$	False	
$f(t_1,\ldots,t_n)$	Function application	The result of applying function f to $(t_1,\ldots,t_n)$
$R(t_1,\ldots,t_n)$	Predicate test	$(t_1,\ldots,t_n)$ belongs to relation R or predicate R holds for $(t_1,\ldots,t_n)$
$\neg\varphi$	Not	φ is not valid
$\varphi \wedge \psi$	And Conjunction	φ and ψ are both valid
$\varphi \vee \psi$	Or Disjunction	Either φ or ψ are valid
$\varphi \rightarrow \psi$	Implication	Whenever φ is valid ψ must also be valid
$\forall x.\varphi$	All Universal	φ is valid for all x in the domain
$\exists x.\varphi$	Some Existential	φ is valid for some x in the domain

Atomic formulas (besides the constants denoting true and false) are predicate tests, i.e., checking if a tuple of terms satisfies the predicate (belongs to the relation). Such atomic formulas can be combined into more complex formulas using the standard boolean connectives already present in PL. However, in FOL we can also introduce *variables* by quantifying, existentially or universally, over the all universe. Using quantifiers we can express properties about the domain in a very compact way.

[18] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[19] https://en.wikipedia.org/wiki/Floyd–Warshall_algorithm

For example, we can express some basic well-formedness referential integrity properties about our problem domain using universal quantification as follows.

$\varphi_1 \stackrel{\text{def}}{=} \forall$`x.`$\forall$`y.(routeEntry(x,y)` $\rightarrow$ `route(x)` $\wedge$ `signal(y))`
$\varphi_2 \stackrel{\text{def}}{=} \forall$`x.`$\forall$`y.(hasTvdSection(x,y)` $\rightarrow$ `route(x)` $\wedge$ `tvdSection(y))`

Property φ_1 asserts that for every pair `(x,y)` belonging to `routeEntry`, `x` must be a route and `y` must be a signal. Property φ_2 is very similar, asserting that `hasTvdSection` only relates routes to TVD sections.

Using a combination of universal and existential quantification we can specify our first desired safety property, namely that every signal is an entry point of a route.

$\varphi_3 \stackrel{\text{def}}{=} \forall$`x.(signal(x)` $\rightarrow$ $\exists$`y.(route(y)` $\wedge$ `routeEntry(y,x)))`

Using function `distance` we can specify our second safety property, namely that every signal is at least 20 meters away from the next switch.

$\varphi_4 \stackrel{\text{def}}{=} \forall$`x.`$\forall$`y.(signal(x)` $\wedge$ `switch(y)` $\rightarrow$ `distance(x,y)≥20)`

This property seems stronger than what was expressed in natural language, since it requires that every signal is at least 20m away from any switch, but it actually is equivalent, since the next switch is the closest one, and if that switch is at least 20m away then all other switches are also.

A concrete valuation for all the predicates and functions for a particular universe is known as a *first-order structure* or *model* in FOL. The fact that formula φ holds (is satisfied) in model $\mathcal{M}$ is denoted by $\mathcal{M} \models \varphi$. Above we presented a concrete example of a FOL model extracted from a railML model. An essential task in this project is precisely to check for a concrete model $\mathcal{M}$ if $\mathcal{M} \models \varphi$ holds. Due to the quantifiers and the fact that the universe is potentially infinite (as is the case in our example model, due to the inclusion of natural numbers), this will not always be possible. However if all quantifiers are *bounded* by a finite predicate it is trivial to implement that check. A universal quantifier is bounded by a predicate `P` if it is of the shape $\forall$`x.(P(x)` $\rightarrow$ φ`)`, that is, we are only interested in checking φ for values of `x` that belong to `P`. An existential quantifier is bounded by a predicate `P` if it is of the shape $\exists$`x.(P(x)` $\wedge$ φ`)`. In all the examples of formulas given so far the quantifiers are bounded, so it would be trivial to check their validity for a particular model. In fact, all of them are valid for the concrete model presented above, with the exception of φ_4, which is false since `distance(x,y)≥20` is not true for signal S2 and switch P2. To highlight the fact that quantifiers are bounded it is common to use a special syntax, for example, writing $\forall$`x:P.`ϕ instead of $\forall$`x.(P(x)` $\rightarrow$ φ`)` and $\exists$`x:P.`ϕ instead of $\exists$`x.(P(x)` $\wedge$ φ`)`. With this alternative syntax our two safety requirements could be specified more succinctly as follows.

$\varphi_3 \overset{\text{def}}{=} \forall \texttt{x:signal.}\ \exists \texttt{y:route. routeEntry(y,x)}$
$\varphi_4 \overset{\text{def}}{=} \forall \texttt{x:signal.}\ \forall \texttt{y:switch. distance(x,y)} \geq 20$

A more complex problem is checking whether a formula is logically *valid*, that is, checking that it is satisfied by any possible model. This is a classic undecidable problem, since there is no algorithm that can, in general, provide that answer. However, checking the validity of a formula can also be relevant for this project, in particular to support the task of rule design, i.e., finding the correct specification for natural language requirements. For example, we could wish to check if for all models satisfying well-formedness properties, such as φ_1 or φ_2, a particular rule is redundant or can be specified more succinctly. For example, we might want to check if rule φ_3 could be specified alternatively as follows.

$\varphi_6 \overset{\text{def}}{=} \forall \texttt{x.(signal(x)} \rightarrow \exists \texttt{y.routeEntry(y,x))}$

To determine this we would need to check the validity of the formula $(\varphi_1 \land \varphi_2) \rightarrow (\varphi_3 \leftrightarrow \varphi_5)$, specifying that the two formulations are equivalent in all models that satisfy the well-formedness constraints. Validity checking can be made decidable by restricting the logic to a well-behaved subset or by just restricting the universe to be finite. A well known tool for formal structural design that follows the latter approach is Alloy[20]. Instead of restricting the logic, another option is to use techniques that may simply fail to provide an answer if the formula falls outside a decidable fragment of the logic. For example, most modern SMT solvers, such as Z3[21], can check the validity of many first-order formulas, but may return an *unknown* in some cases.

Temporal logic

While FOL allows us to specify properties about a fixed interpretation of predicates and functions, Temporal Logic (TL) allows us to specify properties about mutable predicates and functions, whose value changes over time, as the system evolves from state to state. Many different variants of TL exist, differing, for example, on how they formalize the behaviour of a system or how they register the passage of time.

Concerning behaviour there are two main options: *linear time* vs *branching time*. With linear time, the behaviour of a system is formalized by its set of execution *traces*, being a trace an infinite sequence of states. With branching time, the behavior of a system is formalized by its set of *computation trees*, containing each all reachable states from a possible initial state of the system. A computation tree also records for each state all possible choices for next states, and thus in a branching time TL we can specify properties about possible (but not inevitable) sequences of events, such as whether it is always possible to return to an initial state. A trace does not record such choices, and thus with a linear time TL it is not possible to specify such properties. Surprisingly there are some properties that can only be specified with linear time, namely properties related to execution fairness, requiring events that are somehow continuously

[20] http://alloytools.org
[21] https://en.wikipedia.org/wiki/Z3_Theorem_Prover

enabled to eventually occur. Verification of such properties in branching time logics requires special purpose algorithms.

Concerning the passage of time, we can have *timed* or *untimed* logics. In the former, the actual clock time of events is registered, and it is possible to express properties that impose real-time constraints on the occurrence of events, such as specifying that something will inevitably happen within a given time limit. In untimed logics it is only possible to specify properties about the order of events, such as causality constraints.

The most popular (and easy to use) variant of TL is the *Linear Temporal Logic* (LTL)[22]. LTL is an untimed linear time logic that has several temporal operators to specify properties about the execution traces of the system. These operators are listed in the following table, which also presents their informal semantics.

Syntax	**Name(s)**	**Informal semantics**
G φ □φ	Always Globally	φ is always valid
F φ ◆φ	Eventually Finally	φ will eventually be valid
X φ ○φ	neXt After	φ is valid in the next state
φ **U** ψ	Until	ψ will eventually be valid and φ is valid until then
ψ **R** φ	Releases	φ can only stop being valid after ψ becomes valid

The most well known temporal operators are **G** (always) and **F** (eventually), which allow us to quantify universally or existentially over the states of a trace, respectively. In particular, LTL formula **G** φ requires φ to be true in all the states of all possible execution traces, allowing the specification of simple *safety* properties, such as invariants, that prevent "bad things" from happening. On the other hand **F** φ requires φ to be true in some state of all possible execution traces, allowing the specification of *liveness* properties, forcing the system to progress and achieve "good things". Temporal operators can be nested, as in formula **G** (φ → **F** ψ). In those cases, any temporal operator in the inner formula should be evaluated in the trace suffix starting at any state where the inner formula is required to hold. In this particular example, **F** ψ must hold in all trace suffixes starting at states that satisfy φ, meaning that the formula requires that every occurrence of φ must be followed (possibly immediately) by an occurrence of ψ (ψ is a response to φ). Another example of a nested temporal operator formula is **F** (**G** φ), which enforces all execution traces to reach a point where φ is always valid afterwards.

[22] https://en.wikipedia.org/wiki/Linear_temporal_logic

LTL logic is propositional, in the sense that atomic formulas are just propositions of PL. Such logic is not adequate to express temporal properties over complex domains, as is the case of railway networks. A better alternative is First-order LTL (FOLTL), a logic that combines FOL with LTL. A FOLTL formula is evaluated in a trace, an infinite sequence of states where each state is a first-order structure containing a valuation for all predicates and functions, with the special case that immutable predicates and functions have the same valuation in all states. The value of immutable predicates and functions is sometimes known as a *configuration* of the system.

For example, to express our third safety requirement we could consider all predicates and functions presented above as immutable, whose value defines a particular configuration of the railway network. In addition we could consider three mutable unary predicates: `closed`, containing the set of signals whose aspect is closed in a particular state; `proceed`, containing the set of signals whose aspect is proceed in a particular state; and `vacant`, containing the set of TVD sections that are vacant in a particular state. A (not realistic) trace of the system could be the following, with 5 distinct states, where some train repeatedly traverses the route from the upper left buffer stop to the lower right border, crossing TVD sections `TS3`, `TS1`, and `TS2`, in this order.

$$\begin{tikzcd}
0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[lll, bend right]
\end{tikzcd}$$

Since this (infinite) trace repeats itself it is possible to represent it finitetly, by a sequence of 5 states with a mark in the second state signaling that it is the state that succeeds the last one. The value of the three mutable predicates in the five distinct states is the following, where the value of each predicate in a particular state is signaled by adding the state identifier as a subscript to the predicate name. The point where reentry state after looping back is also signaled.

```
closed_0   = {(S1),(S2),(S3)}
proceed_0  = {}
vacant_0   = {(TS1),(TS2),(TS3),(TS4)}
------ loop starts here
closed_1   = {(S1),(S2),(S3)}
proceed_1  = {}
vacant_1   = {(TS1),(TS2),(TS4)}
------
closed_2   = {(S2),(S3)}
proceed_2  = {(S1)}
vacant_2   = {(TS1),(TS2),(TS4)}
------
closed_3   = {(S1),(S2),(S3)}
proceed_3  = {}
vacant_3   = {(TS2),(TS3),(TS4)}
```

```
------
closed₄   = {(S1),(S2),(S3)}
proceed₄  = {}
vacant₄   = {(TS1),(TS3),(TS4)}
```

Equipped with these mutable relations we can use the temporal *always* operator to express some basic invariants, that ensure that they only contain entities of the appropriate type.

```
G ∀x.(closed(x) → signal(x))
G ∀x.(proceed(x) → signal(x))
G ∀x.(vacant(x) → tvdSection(x))
```

We can also express a more fundamental safety property of our system, requiring that signals cannot exhibit aspects *closed* and *proceed* at the same time.

```
G ∀x.(¬closed(x) ∨ ¬proceed(x))
```

By combining the always and releases temporal operators we can express our third example safety requirement, namely that the entry signal of a route can only change its aspect from *closed* to *proceed* if all TVD sections along the route are *vacant*. The releases operator is very useful to express that an event can only occur after another. In particular, ψ **R** $\neg\varphi$ forces φ to remain false until ψ becomes true, meaning that φ can only be valid after ψ becomes valid. If ψ is never valid then φ can never become valid also. Using **G** and **R** our requirement can be specified as follows.

```
G ∀x.∀y.(routeEntry(x,y) ∧ closed(y) →
         (∀z.(hasTvdSection(x,z) → vacant(z)) R ¬proceed(y)))
```

Given the outermost always operator, the formula in the RHS of the implication is required to hold in every state the formula on the LHS holds, that in every state where the entry signal y of a route x is closed. When this trigger condition happens we want to force signal y to only have aspect *proceed* after all TVD sections in route x are vacant. For this we use the **R** operator as described above, with `¬proceed(y)` on the RHS and the release condition `∀z.(hasTvdSection(x,z) → vacant(z))`, requiring every TVD section z that belongs to x to be vacant.

As we can see from this example, translating a natural language requirement to TL is not an easy task. As such, some research has been conducted in developing so-called *temporal specification patterns*, catalogues of formula patterns that capture the most typical constraints occuring in real systems. The most well-known specification pattern catalogue was developed by Matthew B. Dwyer (and others)[23] and is publicly available online[24]. This catalogue is divided between *occurrence* patterns, talking about the occurrence of a given event or state in an execution trace, and *order* patterns, talking about the relative or causality order between events. Among the former we have, for example, patterns such as absence (something never happens) or existence (something will happen). Among the latter we have, for example, the above mentioned *response* pattern. Such patterns can be combined with *scopes*, stating when is the pattern expected to occur. Possible scopes are *globally*, the pattern is always valid, *after* some event or state, or *between* occurrences of specific pairs of states or events. For each combination of pattern and scope the catalogue details how it can be specified in a variety of temporal logics, including LTL.

The key verification technique for TL is *Model Checking* (MC)[25]. The goal of a *model checker* is to automatically verify if a TL formula is valid in a finite *model* of a system. The term model is used with many different meanings, even in the context of formal specification and verification. Above we presented the term model to denote a first order structure where a FOL formula can be evaluated. In MC a model of a system is a formal description of how it can evolve over time, typically some sort of finite transition system, where states are valuations to the system variables and transitions correspond to occurrences of events. Different model checkers allow us to describe this transition system in different ways. For example, in NuSMV[26] it is described more or less explicitly using a DSL, by stating for each variable how its next state value is computed from the one of the current state, while in TLC[27] or Electrum[28] it is described declaratively, by a temporal formula that specifies when events can occur between consecutive states. All these model checkers support the specification of the desired properties using a linear time TL, the standard propositional LTL in the case of NuSMV, the *Temporal Logic of Actions* (TLA) in the case of TLC, and a temporal extension of Alloy in the case of Electrum. The latter two are variants of FOLTL. NuSMV also allows the specification of temporal properties in a branching time TL, namely the *Computation Tree Logic* (CTL)[29].

The model checking procedure tries to refute the formula, that is find a counterexample that proves that it is not valid. If such counterexample cannot be found then the formula is valid. Model checking can be *explicit* or *symbolic*. In the former the transition system is explicitly represented as a graph in memory, and model checking is done by resorting to graph algorithms such as traversals. For example, to model check `G φ` all reachable states of the graph are traversed while trying to find a state where φ is not valid. If such a state is reached, a counterexample trace is output, corresponding to the path from an initial state to the culprit

[23] Patterns in Property Specifications for Finite-state Verification, *Matthew B. Dwyer, George S. Avrunin and James C. Corbett* in Proceedings of the 21st International Conference on Software Engineering, May, 1999.
[24] https://matthewbdwyer.github.io/psp/
[25] https://en.wikipedia.org/wiki/Model_checking
[26] https://nusmv.fbk.eu
[27] https://lamport.azurewebsites.net/tla/tla.html
[28] http://haslab.github.io/Electrum/
[29] https://en.wikipedia.org/wiki/Computation_tree_logic

state. If no conter-example is found then the property is valid. Of course, such a procedure is only complete if the transition system is finite.
The TLA model checker for TLA is an example of an explicit state model checker. Symbolic model checkers represent the transition system and execute reachability algorithms symbolically, by representing sets of states and transitions by formulas, and can thus better avoid the state-explosion problem that can hamper explicit model checkers. Both NuSMV and Electrum are symbolic model checkers. Most of the model checkers also offer *Bounded Model Checking* (BMC) verification procedures, where the transition system is only explored up to a given depth when trying to find counterexamples. In this case, absence of a counterexample no longer entails the validity of the formula since counter-examples may exist beyond the set bound, but BMC is typically much more efficient than the standard complete MC procedure, and is commonly used in the early phases of the formalization process to debug problems in the specification of the system model or in the specification of the desired properties. Once enough confidence is established by using BMC, complete MC should be used for the final verification.

Tools and techniques for railML® verification

As briefly presented in the introduction, formal methods have been widely and successfully applied to the railway domain. However, verification techniques that support rich topological models, which are the focus of this project, are scarce. This section surveys such techniques, focusing on techniques that support railML's topological model.

Schema validation

The railML organization provides its own tool for the visualization and validation of railML models, railVIVID[30]. As of the latest beta version from 2019, this tool also supports the validation (but not visualization) of railML 3.1 models. However, railVIVID focuses essentially on validating the XML documents against the railML schema and checking referential integrity, such as validating whether identifier references point to existing elements. Although this is not even close to the type of properties that we intend to verify in DigiLightRail, it could still be useful to integrate railVIVID in the proposed toolchain, so that tools down the line have some guarantees about the correction of the input railML models and can ignore such validations.

Infrastructure verification

As far as we are aware of, the only existing work for verifying the static aspects of railway design is the one developed in the context of the Norwegian RailCons - Automated Methods and Tools for Ensuring Consistency of Railway Designs - project[31], part of which has been integrated and commercialized in the RailCOMPLETE framework[32]. Their approach to verification, best summarized in a 2018 scientific journal publication[33], has the goal of

[30] https://www.railml.org/en/user/railvivid.html
[31] https://www.mn.uio.no/ifi/english/research/projects/railcons/
[32] https://www.railcomplete.com
[33] Bjørnar Luteberget, Christian Johansen: Efficient verification of railway infrastructure designs against standard regulations. Formal Methods Syst. Des. 52(1): 1-32 (2018)

continuously verifying compliance with regulations during the CAD-centered design process, focusing on rules addressing the station layout and interlocking.

In this approach, from the user perspective, both the civil, track, and signal design is performed in AutoCAD, supported by a plug-in that deploys the verification procedures and reports back counter-examples, presenting them in the CAD drawing. The RailCOMPLETE plug-in implements what the authors dub *semantic CAD*. The designers are provided with a block library that defines a block for each relevant railway infrastructure object (such as signals, detectors or switches), so that the resulting CAD not only contains geometrical information, but also high-level railway concepts. From these, the plug-in continuously generates a railML representation of the topological information and assigns railML fragments to each block reference and relevant geometrical object (such as polylines representing tracks) using DWG's *extended dictionaries*. Additionally, railway information that is not local to any object (such as interlocking routes) is stored in the DWG's *global* extension dictionary.

Although railML 3.1 can currently store most topology and interlocking information, at the time of publication, railML (then at version 2) was not sufficiently expressive to encode some critical interlocking information, namely the conditions for a route to be active, such as required switch positions, detection sections that must be empty, and conflicting routes[34]. To fill that gap, M. Bosschaart et al.[35] proposed the formalization of an interchangeable data format for interlocking based on railML. An extension to railML 2 was proposed - which still did not comply with the RTM topological model - to represent interlocking information, which does not seem fully compatible with what would become the interlocking subschema of railML 3.1. RailCOMPLETTE stores interlocking information in this XML format for the route-based tabular specification of interlockings. The goal of having this information is not to verify dynamic interlocking aspects, but rather rules regarding the assignment of those conditions (e.g., if the route crosses a detection section, it must have a corresponding activation condition).

The actual verification engine is implemented in Datalog, a subset of the Prolog logic programming language supporting a fragment of FOL that includes existential quantifications. Facts of the knowledge-based are inferred from the railML embedded in the DWG file. Derived facts are defined for commonly used concepts to ease the definition of rules, including reachability and distance predicates. To that purpose, the selected Prolog implementation supports recursive definitions and arithmetic operations. The approach also explores incremental verification features of Prolog for improved efficiency when an already verified design is updated by the designers.

The rules provided as examples were selected from the Norwegian infrastructure manager's regulations. These include layout rules (such as "a signal must be placed before the first facing switch of a station", "detectors must be placed within a certain distance of each other", or "station boundaries must be preceded by a single exit signal") and interlocking rules (such as "a

[34] As described above, in version 3.1 of railML this information is covered by the interlocking sub-schema.
[35] Mark Bosschaart, Egidio Quaglietta, Bob Janssen, Rob M. P. Goverde: *Efficient formalization of railway interlocking data in RailML.* Inf. Syst. 49: 126-141 (2015)

route must have entry and exit signals” or “the route must have an activation condition for each detection section if crosses”).

By being CAD-centric, this approach did not expect any particular expertise from the designers, except for the definition of rules that would be defined in Datalog by an expert. Later, the same authors designed a (preliminary) DSL for the specification of rules for railway infrastructures[36]. The goal was to both allow railway engineers to participate in the creation and changing rules, but also support designers in interpreting the presented counter-examples. The result of this research was RailCNL, a controlled natural language. Such language is built on top of the Grammatical Framework programming language by defining an abstract syntax and then a concrete syntax supported by a resource grammar library for natural languages. Additionally, RailCNL rules can be annotated with traceability to the original natural language requirement to promote the understandability of the rules and counter-examples. On average, they were able to encode in RailCNL 74% of the Norwegian regulations relevant for railway infrastructure design.

As an example, the following natural language regulation

“*No detection section shall be shorter than 21 meters.*”

would be written in RailCNL as

“*The distance from an axle counter to another must be greater than 21.0m.*”

which corresponds the following abstract syntax, to be subsequently translated into Datalog for verification

```
DistanceRestriction Obligation
       (SubjectClass (StringClassNoAdjective (StringClassMasculine "axle_counter")))
       (AnyFound (AnyDirectionObject SubjectOtherImplied))
       (Gt (MkValue (StringTerm "21.0m")))
```

Interlocking verification

While the goal of railML is not to encode the dynamic aspects of a railway system, it would be desirable to have techniques and tools for the verification of dynamic interlocking properties (such as the third safety property above) that take into account the infrastructural aspects of the network described in railML models. Unfortunately, as far as we are aware only the work by T. Gonschorek et al.[37] proposed such a technique, albeit with many limitations. In particular, they proposed a formalization of railway infrastructures and interlocking systems independent of the target model checkers, and show that the approach can be instantiated to several of them. The focus is on achieving a tool-independent technique, and railML 2 can be used as the input language and translated to this formalism. Unfortunately, it does not use the interlocking

[36] Bjørnar Luteberget, John J. Camilleri, Christian Johansen, Gerardo Schneider: Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. SEFM 2017: 87-103
[37] T. Gonschorek, L. Bedau & F. 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*. CRC Press, 2018.

subschema of railML which at the time was still under development. Thus, for interlocking information they resort to the tabular representation, using CSV files. Moreover, the approach still does not support the RTM-compliant topological model released with railML 3.1.

A route is represented by a set of track segments (`<netElement>`s in railML 3.1), switches, and signals (no other elements were considered). Train movement is abstracted, not taking into consideration velocity or the length of each track segment. A train is just a set with one or more adjacent track segments. Verification considers up to two trains per route, to model possible collisions. Only two aspects are considered per signal (*red* and *green*). The interlocking control system seems to be very abstract, in the sense that no concrete locking algorithm or control logic is verified, and assuming that locking a route just locks all the constituent elements.

Several generic safety properties are verified, whose specific instantiations are automatically derived from the topology and route models:

- Absence of train collision;
- Absence of derailment at switches (for example, because they are not set to the direction the train is approaching);
- Flanks are protected (switches leading to the route are in a state that makes trains deviate from the route).

The network and the expected properties are specified in the *System Analysis Modeling Language* (SAML)[38], supported by the tool *Verification Environment for Critical Systems* (VECS)[39], which verifies SAML models with several third party model checkers (including probabilistic ones such as PRISM[40]). In this work they experimented with nuXmv[41] (in complete mode), iimc[42] (both in complete IC3 mode and BMC mode), and aigbmc[43] (BMC mode only), for verifying the above properties in an example of a real station for which they were expected to hold. To verify the performance when detecting counter-examples, some faults were injected in the model. For valid properties iimc with IC3 performed the best, but when counter-examples were present iimc with BMC was preferable, with the exception of a faulty flank protection, whose counter-example required many steps, and for which all BMC engines failed to terminate.

[38] Gudemann, Matthias, and Frank Ortmeier. "A framework for qualitative and quantitative formal model-based safety analysis." In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pp. 132-141. IEEE, 2010.
[39] https://cse.cs.ovgu.de/vecs/index.php/product/vecs
[40] https://www.prismmodelchecker.org
[41] https://nuxmv.fbk.eu
[42] https://github.com/mgudemann/iimc
[43] http://fmv.jku.at/aiger/

SCADE?

Simulation?

https://www.witpress.com/elibrary/wit-transactions-on-the-built-environment/74/12035

Reverse engineering?

https://www.researchgate.net/profile/Tim-Gonschorek/publication/326032697_Bringing_formal_methods_on_the_rail_On_automatic_verifying_railroad_interlockings_from_railML_models/links/5b34a45f0f7e9b0df5d2f7d7/Bringing-formal-methods-on-the-rail-On-automatic-verifying-railroad-interlockings-from-railML-models.pdf

Deliverable description

Pesquisa, identificação de linguagens formais de aplicação a ferramentas de modelação e engenharia [entregável: 1 relatório] A ferramenta a desenvolver tem duas vertentes principais: uma diz respeito às traduções entre as diferentes linguagens, e outra à validação e verificação de modelos railML3.0. Esta tarefa tem como objectivo explorar as técnicas formais mais adequadas a este domínio de aplicação em concreto, tendo em consideração a expressividade das linguagens e as classes de propriedades a verificar.

RailML side of DigiLightRail

RailML wikis

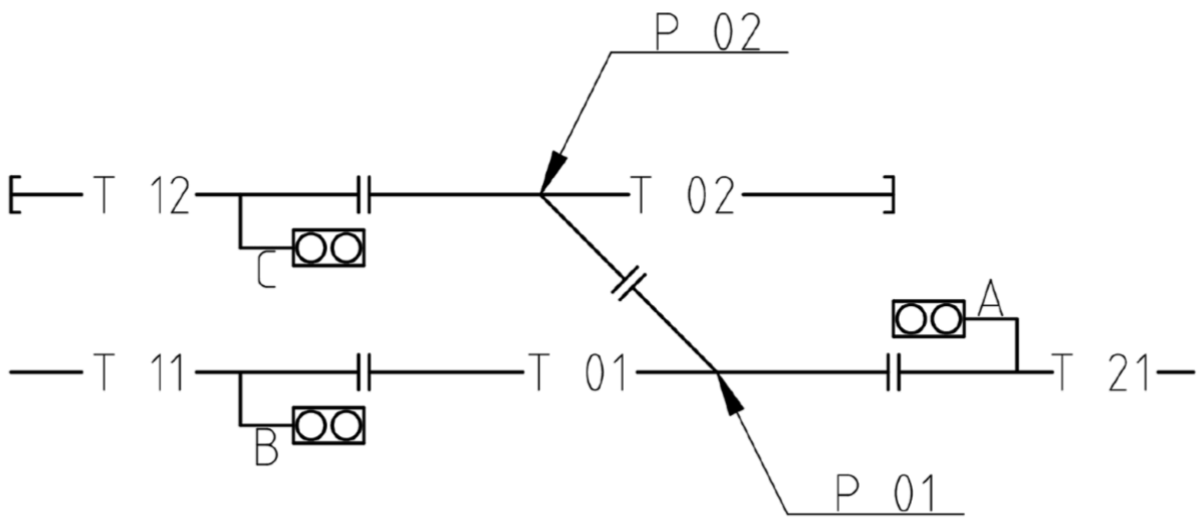
- Version 2: https://wiki2.railml.org/wiki/Main_Page
- Version 3.0 https://wiki3.railml.org/wiki/Main_Page where one can find:

> **Interlocking (IL)**
>
> →*Main Article: Interlocking*
>
> The interlocking subschema will focus on information that infrastructure managers and signal manufacturing industry typically maintain in signal plans and route locking tables:
>
> - Data transfer: a standard data exchange format will allow the automation of data transfer, which is the process of adapting a railway interlocking and signaling system to a specific yard.
> - Simulation programs: the railML® IL schema allows modelers to quickly absorb information about the interlocking systems such as timing behavior and routes and analyze the impact on railway capacity.
>
> The subschema is **currently under development** with the participation of some European infrastructure managers and the signal manufacturing industry based on railML 3 and UIC's RailTopoModel *(external link)*. A list of use cases for railML's interlocking scheme can be found here IL:UseCases.

The main element is IL:interlocking.

Link to RTM: http://www.railtopomodel.org/en/state-of-development.html

Example: “hello world” of railML track topology



Abstract model of corresponding railML below (NB: *Xs* means *X**):

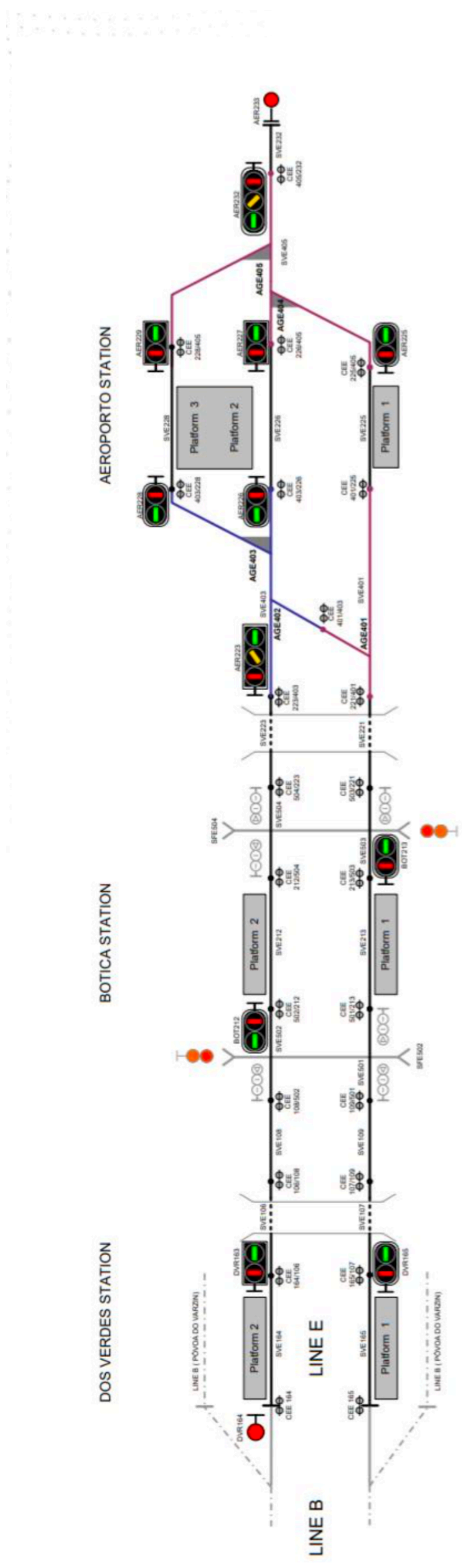
Infrastructure = Tracks
Track = Topologys x Elements x Ocs
Topology = EndPoint x EndPoint x Connections + Cross + …
Element = Edge + ...
Oc = Signal + Detector + …
Connection = Switch + Sequential
Switch = SId x SId x Pos x Orientation
EndPoint = EPId x Pos x (1 + Stop)
Sequential = EPid x EPId
Signal = SId x Pos x Dir x Type x (1 + Speed)

Invariants: is the *Sequential* relation (etc) symmetric? CF. two ways

XML on-line browser: https://countwordsfree.com/xmlviewer

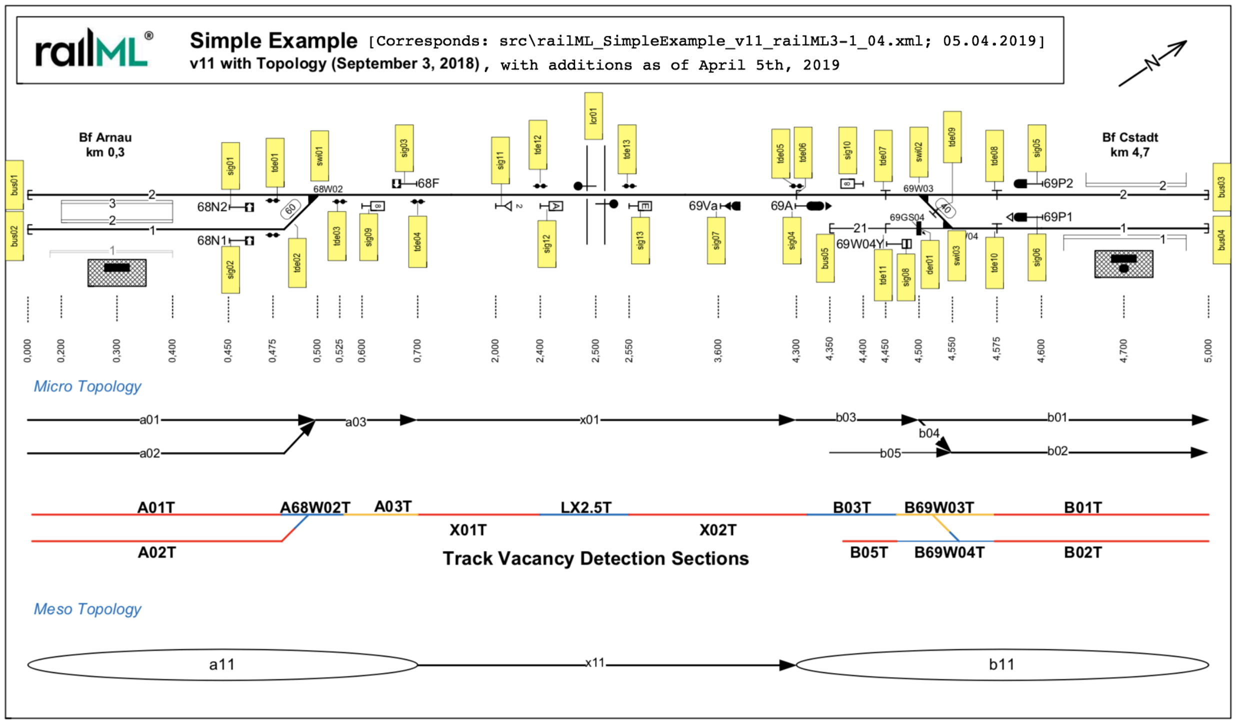
```
<railml>
<infrastruxcture id='example'>
<tracks>
① <track id='T12'>
 <trackTopology>
  <trackBegin id='T12a' pos='0'>
②  <connection id='T12ac' ref='T02bc'/>
  </trackBegin>
  <trackEnd id='T12b' pos='42'>
②  <bufferStop id='T12bc'/>
  </trackEnd>
 </trackTopology>
 <ocsElements>
  <signals>
③  <signal id='SigC' pos='10' dir='down' type='main'
     ↪ />
  </signals>
 </ocsElements>
</track>
① <track id='T02'>
 <trackTopology>
  <connections>
④  <switch id='P02' pos='21'>
    <connection id ='P02c' ref='P01c' orientation='
        ↪ incomming'/>
   </switch>
  </connections>
  <trackBegin id='T02a' pos='0'>
②  <bufferstop id='T01ac'/>
  </trackBegin>
  <trackEnd id='T02b' pos='42'>...</trackEnd>
 </trackTopology>
</track>
① <track id='T01'>
 <trackTopology>
  <connections>
④  <switch id='P01' pos='21'>
    <connection id ='P01c' ref='P02c' orientation='
        ↪ outgoing'/>
   </switch>
  </connections>
  <trackBegin id='T01a' pos='0'>
②  <connection id='T01ac' ref='T21bc'/>
  </trackBegin>
  <trackEnd id='T01b' pos='42'>...</trackEnd>
 </trackTopology>
</track>
</tracks>
</infrastructure>
</railml>
```

EFACEC Hello World:

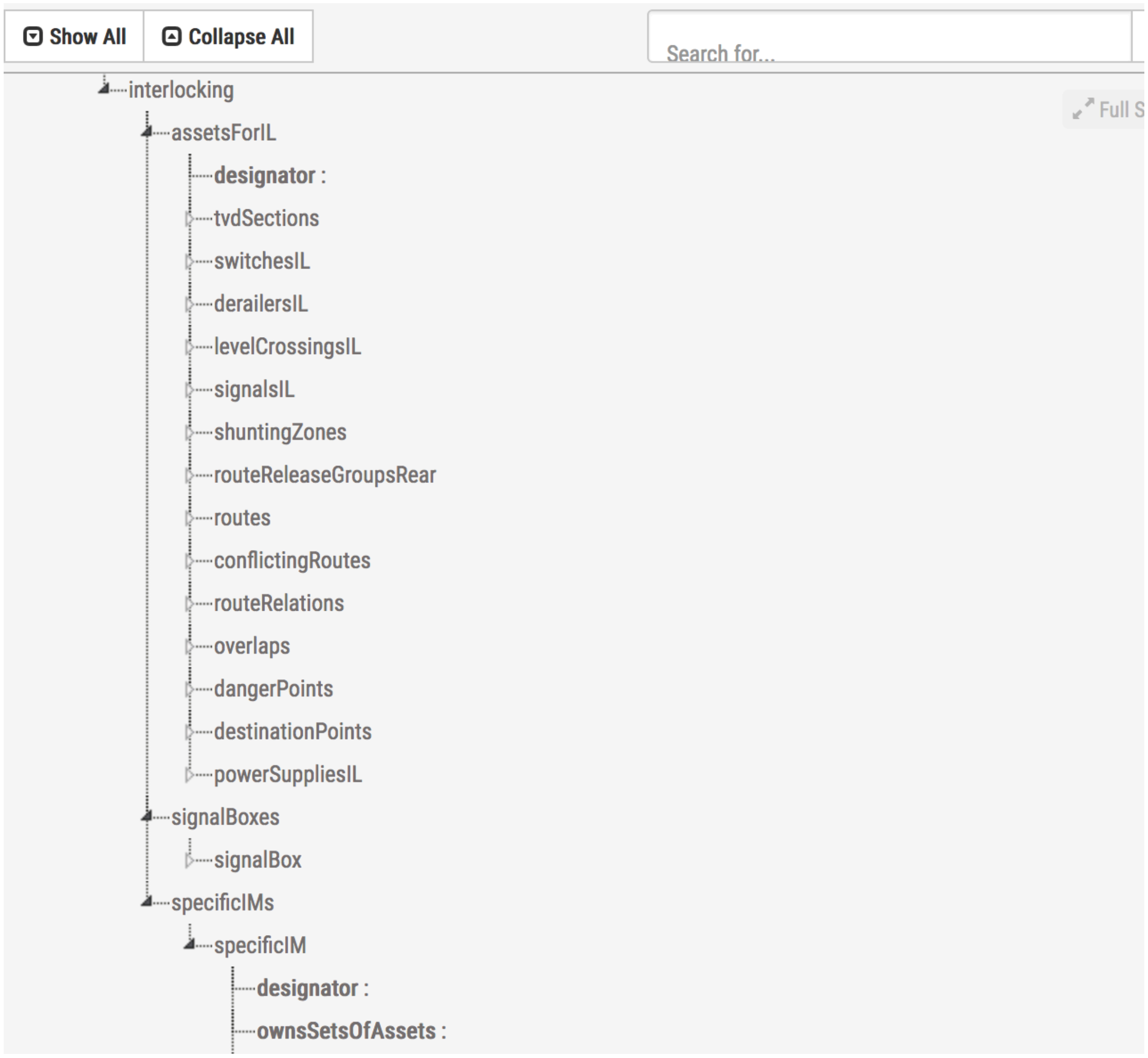


Realistic railml 3 example taken from: https://www.railml.org/en/user/exampledata.html

Visualization first:



Simple Example [Corresponds: src\railML_SimpleExample_v11_railML3-1_04.xml; 05.04.2019]
v11 with Topology (September 3, 2018), with additions as of April 5th, 2019
Bf Arnau km 0,3
Bf Cstadt km 4,7
Micro Topology
Track Vacancy Detection Sections
Meso Topology
a11
x11
b11
A01T
A68W02T
A03T
LX2.5T
X01T
X02T
B03T
B69W03T
B01T
A02T
B05T
B69W04T
B02T



- interlocking
  - assetsForIL
    - **designator :**
    - tvdSections
    - switchesIL
    - derailersIL
    - levelCrossingsIL
    - signalsIL
    - shuntingZones
    - routeReleaseGroupsRear
    - routes
    - conflictingRoutes
    - routeRelations
    - overlaps
    - dangerPoints
    - destinationPoints
    - powerSuppliesIL
  - signalBoxes
    - signalBox
  - specificIMs
    - specificIM
      - **designator :**
      - **ownsSetsOfAssets :**

People

Anne Elisabeth Haxthausen - http://www.imm.dtu.dk/~aeha/

*** ter Beek, Fantechi, FMICS people, Olivier Lecombe (Clearsy) etc ...

Projects

Conferences

Please see pag 189 proc. FMICS 2020 (+ etc)

Cf FMICS 2020, Videos should appear in https://www.youtube.com/channel/UCK9p1Z8nIPTP4Uv5Qodb1og; já lá está o da panel discussion: https://www.youtube.com/watch?v=8kXER8Z01kk

Tools

XML on-line browser: https://countwordsfree.com/xmlviewer

Example: railML 2.4 - https://www.dropbox.com/s/8ytidw52q901w9t/railML_SimpleExample_v11_railML2-4_01.xml?dl=0

https://github.com/luteberget -> rolling https://luteberget.github.io/rollingdocs/rolling.html

. Railway Infrastructure and Layout Aided Designer (https://www.rail-aid.com)

. The ERTMS/ETCS signalling system, http://www.railwaysignalling.eu/wp-content/uploads/2016/09/ERTMS_ETCS_signalling_system_revF.pdf

. Iliasov, A., Lopatkin, I., Romanovsky, A.: The SafeCap Platform for Modelling Railway Safety and Capacity. In: Computer Safety, Reliability, and Security. LNCS, vol. 8153, pp. 130–137. Springer (2013)

. International Union of Railways (UIC): RailTopoModel - Railway infrastructure topological model (2016), ISBN 978-2-7461-2513-1

. RailCons tools (https://www.mn.uio.no/ifi/english/research/projects/railcons/) Christian Johansen, Bjørnar Luteberget RailCons is a research project that has been running for 4 years and will end in August 2019. This project was funded partially the Norwegian Research Council and by the company Railcomplete, where one PhD student was hired to do research on “Automated Methods and Tools for Ensuring Consistency of Railway Designs”. There have been produced five related tools/prototypes, three of these have been already presented at iFM, SEFM, FMCAD (with two Best Paper Awards), the forth one will be presented at FM’19, and the fifth is being submitted to iFM’19. More info exists on the project’s webpage (including videos, posters, papers, slides).

Falam aqui de formal methods e interlocking
https://www.researchgate.net/profile/Tim-Gonschorek/publication/326032697_Bringing_formal_methods_on_the_rail_On_automatic_verifying_railroad_interlockings_from_railML_models/links/5b34a45f0f7e9b0df5d2f7d7/Bringing-formal-methods-on-the-rail-On-automatic-verifying-railroad-interlockings-from-railML-models.pdf

https://ieeexplore.ieee.org/abstract/document/7911872

https://link.springer.com/chapter/10.1007/978-3-030-18744-6_15

https://link.springer.com/chapter/10.1007/978-3-319-33693-0_31

https://link.springer.com/chapter/10.1007/978-3-319-48989-6_49

https://cse.cs.ovgu.de/cse-wordpress/wp-content/uploads/2018/02/Esrel2018_GonschorekEtAl_ModelCheckingRailMLInterlockings.pdf

Open track -- https://mtc-aj.com/library/1146_EN.pdf
…….

(**FMICS 2020 invited talk, extended abstract**) Applying Formal Methods in Industrial Railway Applications at Thales
Stefan Resch, Thales Austria GmbH, Vienna, Austria, stefan.resch@thalesgroup.com, www.thalesgroup.com
The application of formal methods is intended to improve software quality. While common tools that perform static code analysis such as Coverity [3] are well known and applied in the industry, this talk presents three use cases at Thales that leverage formal methods and the according tools to an even larger extent.
The conditions for applying formal method tools for safety critical software in the railway domain are defined by the CENELEC EN 50128 [2] standard. This standard highly recommends the use of formal methods for safety relevant projects for the highest safety integrity levels (SIL) of SIL3 and SIL4. The CENELEC EN 50128 categorizes tools into three different types from T1 to T3 depending on whether they can introduce faults into the safety critical software. Here tools related to formal methods usually are of type T2, since they are used for verification and may fail to identify a fault, but cannot introduce them themselves. This requires that, when used in a safety relevant project, (1) the selection of the tool and its assigned category are justified, (2) potential failures are identified, as well as measures to handle such failures, (3) the tool has a specification or handbook, (4) it is ensured that only justified versions of the tools are used and (5) this justification is also performed when switching versions of the tool.
Each of the use cases presented in the following sections has a different focus, illustrating the vast potential of formal methods. They demonstrate that while formal methods may pose a significant overhead at design time they provide an overall benefit when used in the right context.

2 Use Case: ERTMS Hybrid Level 3

ERTMS Hybrid Level 3 is a concept that enables an increase of track capacity in the railway network by reusing regular signaling and interlocking interfaces to integrate into existing systems while benefiting from the continuous supervision of the trains in network via radio. [4] The specification of this concept was analyzed and validated using a formal model in B [1] and executed at runtime in Pro-B [7]. It was subsequently successfully used in a field demonstration controlling real trains.

3 Use Case: Checking ETCS Level 1 Line Side Data[44]

One of the challenges when deploying the new European Train Control System (ETCS) lines lies in the complexity of the configuration data. We use the tool Emerald for checking the ETCS Level 1 lineside configuration data against rules derived from a customer specific “Book of Rules”. Emerald internally uses B and Pro-B and is developed and maintained by Thales, since it is a highly specific application. The advantage of Emerald’s approach is that many data preparation errors can be caught early on during development, before starting the verification phase of the projects. This tool is actively being used in the current projects.

4 Use Case: TAS Control Plaform

The method of model-checking was used to model and develop fault-tolerant and safety-critical modules for TAS Control Platform, a platform for railway control applications up to safety integrity level (SIL) 4. [8] By model-checking modules in TLA+ [5] and PlusCal [6] core safety and liveness properties of a distributed fault-tolerant protocol were analyzed. A translator from PlusCal to C bridges the gap between model and code.

[44] https://www.ertms.net/wp-content/uploads/2018/10/3-ERTMS-Levels1.pdf