Automatic Model-based Verification of Railway Interlocking Systems using Model Checking

Tim Gonschorek & Ludwig Bedau & Frank Ortmeier
Chair of Software Engineering, Faculty of Computer Science
Otto-von-Guericke-University Magdeburg, Germany

ABSTRACT: The theoretic foundations for formally verifying railway interlocking systems have already been studied extensively. There exist a lot of work covering the application of methodologies like model checking in this context. However, some design faults still remain undetected until final on-track evaluation of the system. This is strongly related to missing automation solutions for real-world models and standards as well as the high theoretical expertise required. There exist many well-developed tools each requiring different modeling formalisms and focusing on a different question/scenario. Without specific experience in formal system modeling, it is extremely complicated to model such complex systems. In this paper, we present a methodology for the automatic model generation and verification of railway interlockings in a tool-independent(!) way. Therefore, we define a generic template set of atomic track elements and safety properties in a formal modeling language applicable with precise semantics. This generic template enables us to verify the structure of any given track layout. The already existing tool support of VECS allows to automatically translate these specifications into various model checkers for verification. More important, we present a robust transformation of the upcoming data exchange format for railway interlocking systems railML into the presented specification template. As a consequence, this approach really may help to bridge the gap between formal methods and system design in railway interlockings. We evaluate this approach on a real-world case studies train station of Brain l’Alleud. We also show the tool-independent modeling by automatically translating the specification to different verification engines and compare their performance.

1 MODEL-BASED VERIFICATION AND INTERLOCKING SYSTEMS

Designing interlocking systems for large railway stations is a very complex task. Lots of different routes, not only for passengers but also for logistics and freight traffic, must be combined with a vast traffic network. Moreover, such railway components, e.g., the interlocking system structure and corresponding route network, are safety critical infrastructures. This means, errors in the network plan, as an unconnected or a dead-end track, an incorrect working switch or wrong scheduled and therefore crossing routes can lead to costly and dangerous hazards. Verifying such railroad interlocking systems by applying formal verification techniques (e.g., model checking (Clarke et al. 1999)) increases the quality a lot.

Even though there exists a vast amount of work on the formal verification of railway interlocking systems, techniques as model checking or deductive verification are not commonly used in practice. We think that one major problem is that there exists no off-the-shelf implementation that can be applied as a simple addon to a given interlocking design tool.

In particular, there exist several tools providing their domain specific language or an interface to a particular modeling tool that can do the verification task. However, either you need to find the one tool that applies to your specific modeling environment, or you must transform your model into the input language of your verification tool. The first approach can lead to the conclusion that there does not already exist such tool and the second is very erroneous and due to the time consumption and required skill level of the engineer.

To overcome this problem, we want to define a structural transformation from an accepted interlocking modeling standard into a formal representation of a railway interlocking system and its routing tables. Over the last ten years, a data scheme standard enabling the interchangeability of railroad design data has been developed by a consortium of leading companies from the interlocking and signaling domain. It is called Railway Markup Language (railML)\(^1\) and

\(^1\)https://wiki.railml.org/index.php?title=Main_Page
defines a data scheme based on XML. By using this standard, it becomes possible to define a standardized verification approach for general interlocking system designs and logical routing tables. Since we think that this standard will be used through industrial application by the next years, a verification tool based on this can help to lower the hurdle of using formal verification techniques for interlocking systems in practice.

In this paper, we define a transformation from a given railML description into a formal model representation which can be used for model-based verification, i.e., for verifying safety properties and additional measures as probabilistic analysis or failure injection techniques. Therefore, we defined a set of template automata for essential interlocking elements and their instantiation with the information derived from the railML representation. Further, we present a real-world case study of the Belgian railway station Braine l’Alleud and a representation of the German train station of the city of Leipzig. This is, to provide an idea of the capabilities enabled by the automatic transformation and corresponding model checking tools.

We think that the fully automatic transformation from an accepted railway data scheme will lower the hurdle for the application of formal verification techniques within the development of interlocking systems and even increases the acceptance of the assessment. This is especially the case since we try to provide a 1:1 transformation being as much information of the original data, which also ensure a full, and easy to understand, traceability from the design to the formal model.

**Related Work** Of course, the verification of railway interlocking systems has already been researched during the last years by several publications as (Cimatti et al. 1998a) or (Haxthausen et al. 2011, Cappart and Schaus 2016, Cimatti et al. 1998b, Bonacchi et al. 2013, Limbrée et al. 2016). These authors showed the overall applicability of formal verification, in particular, model checking, in the context of the analysis of railway interlocking systems. However, (Banci et al. 2004) presented a first general representation of interlocking components using statemate and Harel state charts (Harel 1987). Further, the first transformation of a railway interlocking system from a specific Domain Specific Language into a formal modeling language, Event-B (Abral 2010), was presented by (Iliiasov and Romanovsky 2012) in the context of the SafeCap project (Iliiasov et al. 2013) intending to improve the time-effectiveness of route plans without violating the safety properties of the systems. Therefore, the B verification engine ProB (Leuschel and Butler 2003) was used. However, the main focus of this project was not on the formal verification but the optimization of the system.

The remainder of the paper is the following: In section two we present preliminary background information about the implemented interlocking components and railML. Further, we present the generation algorithm for the formal model by presenting abstractions for the minimal set of required elements (switches, tracks, etc.) and the corresponding safety properties to be verified (derailment, collision, etc.). Section four presents the real-world case study and experimental results of the verification using different model checking tools. The conclusion and further work are given in section five.

## 2 THE BASIC ELEMENTS OF AN INTERLOCKING SYSTEM AND RAILML

Before we introduce the mentioned approach, we need to clear the atomic elements of an interlocking system which must be implemented for a correct abstraction of an interlocking system.

Figure 1 presents a simple example of a railway interlocking system’s track layout (Fig. 1a) as well as the corresponding interlocking table (Fig. 1b) representing the available routes.

The track layout is divided into connected track elements like T01, T02, T11, etc. Track T12, for example, is a dead-end track ending with a bumper. Open-ended track elements T11 and T21 can connect the system to other interlockings. Tracks T01 and T02 are connected by switches (the points P01 and P02) for branching between several track elements.

Points (P01 and P02) can be in nominal position so that trains will drive straight on, or in reverse position so that trains will branch off the current track element. Further, the given interlocking table supports flank protection, i.e., Switch P02 is also switched as a safety function. In normal position, it will discon-
Figure 2: Excerpt from the railML representation of the example track layout.

To prevent wagons from rolling back into the main track, track T12 from the main track T11↔T01↔T21 to the main track T11↔T01↔T21 must be blocked, which signal must be set, or direction of the track.

Railway Markup Language If we want to process the data of a given representation in different steps and tools, we are required to base our transformation on some standard scheme. As mentioned before, we, therefore, focus on the infrastructure definition of railML. The definition of railML in general contains schemes for Timetable Rostering, Infrastructure definitions, Rollingstock, and Interlocking tables and signal plans. It would have also been interesting to use the Interlocking scheme which is meant to support the definition of interlocking and route tables. Unfortunately, this scheme is under development at the moment and therefore not available. Instead, we use a simple csv representation of the route table as given in Fig. 1.

In the following, we present an excerpt of the infrastructure railML representation of the example in Fig. 1. In particular, these are elements covered by route C21 (cf. Fig. 1b), i.e., T12, T02, T01, and correlated elements like switches and signals. In the following, we give a short explanation of the listing in Fig. 2, without providing a complete description of railML. The basic elements of the infrastructure are

3 A TEMPLATE SYSTEM FOR INTERLOCKING SYSTEMS

The basic idea of our transformation system is to provide a set of accepted templates in a formal language, e.g., SAML in our prototype, for single track elements and to derive the instantiations and the links between elements from the railML representations.

The class diagram in Fig. 3 shows the templates and their relations. All infrastructure elements (tracks, signals, switches) are controlled via route automata. Each route holds information about the track segments it contains and the corresponding signal states and switch positions. Therefore, all activities are correlated to the currently active routes, i.e., which tracks must be blocked, which signal must be set, or which switch must be set to a particular position.

Route Scheduling The route scheduling itself is implemented indeterministic, i.e., each nonactive route can request to get active at each time if all required track segments are not blocked by another route. If more than one route tries to get active, we solve this race condition by choosing the route with the lower id (e.g., route 1 before route 2). Overall, more than one route can be active at the same time,
but only one additional route can get active at each time step.

**Logical Trains** At the moment, the train movement is modeled in an indeterministic, logical, way. This means a train can leave a specific track segment or not, without taking into account physical parameters like train velocity and length of the track segment. Each train consists of one to $n$ adjacent track elements to model the movement from one to another track element and trains of different size. If required the formal model can easily extend with physical behavior. Further, we define two trains per route for modeling collisions, flank protection, and other safety properties.

In the following, we present the internal automata of the templates and their basic instantiation idea.

### 3.1 Track

In general, a track can be *clear*, *reserved*, or *occupied*. Since we are verifying the system with two trains, the track can be occupied by train No. 1 or train No. 2 (*occupiedBy1* or *occupiedBy2*). Model checking, in combination with non-deterministic behavior, covers all possible combination of train positions and routes and therefore two trains are sufficient for analyzing all possible train to train situations.

#### railML relation

In the following, we need information on adjacent tracks. This information is derived from the railML element track (cf. Fig. 2 ①) and underlying connection information (cf. Fig. 2 ②). To identify, which tracks are before and behind a track element, we use the railML direction conventions, where direction is from the lower *pos* value to the higher. From this, tracks that are referenced *trackBegin* are precedent tracks and those references at *trackEnd* are subsequent. The assignment of tracks to routes and their required direction (*normal* or *reverse*) is extracted from the route table.

#### Behavior

A track can be requested by a route. This is done indirectly by checking the current state of all routes containing the track.

$$\text{reserveRequest} := \bigvee_{r \in \text{requiringRoutes}} r.\text{state} = \text{commanded}$$

Tracks will be occupied in addition to their neighbor tracks.

$$\text{occupiedContract1} := \text{trackLeft.occupiedBy1} \lor \text{trackRight.occupiedBy1}$$

$$\text{occupiedContract2} := \text{trackLeft.occupiedBy2} \lor \text{trackRight.occupiedBy2}$$

The initial state of the model is with empty tracks, so there has to be a possibility to place some trains on the tracks. Therefor, all possible starting tracks have a non-contradictional formula *occupiedContract0* with

$$\text{occupiedContract0} := \bigvee_{r \in \text{requiringRoutes}} r.\text{commanded}$$

The transition from occupied to clear is enabled when the next track is occupied, and the previous track is clear. This assures that a train will not be split or removed.

Further a train is not forced to enter or leave a track in every step modeling trains with different length.

$$\text{clearContract} :=$$

$$\text{(trackLeft.occupied} \land \text{trackRight.state} = \text{clear}) \lor (\text{trackLeft.state} = \text{clear} \land \text{trackRight.occupied})$$
3.2 Signals

Signals can be in one of two states – stop (red) or proceed (green). States which allow the trains to proceed with low speed are not modeled.

**railML relation** For the signals, we need to know on which track the signal is placed and the direction of the signal, as well as the next track behind the signal to define when the train entered a route and set the signal respectively. These information can be derived from the signal element of the railML description (cf. Fig. 2 3) and the parent track element.

**Behavior** The complete automaton of the signal template is given in Fig. 5. Its initial state is stop. It changes to proceed if the following route (behind this signal) is ready, i.e., the route is accessible. SaveRoute gets true one of the routes starting at the signal is accessible and the signal state changes to proceed.

\[
\text{saveRoute} := \bigvee_{r \in \text{requiringRoutes}} r.\text{proved}
\]

If a train passed the signal, it falls back to stop. Therefore, formula TrainPassed is connected to the track behind the signal to detect when a train passed it. This is detected by a subsequently occupied track.

\[
\text{trainPassed} := \text{trackBehind.isOccupied}
\]

3.3 Switch

A switch can be in two states according to its position: normal (straight on) and reverse (branching).

**railML relation** Besides the id, we must derive the information of the connection relations from the switch, i.e., which tracks are connected via normal and reverse position. This is done by utilizing the switch tag information (cf. Fig. 2 4) and the references to the adjacent track or switch in combination with the parent track element.

**Behavior** A switch automaton has four states, two for each position normal or reverse, and two according to its possibility to change its position. This means a switch can be locked (no change possible), or unlocked (changing the position is possible) for preventing switches from changing while occupied by a train. The transitions normalRequest and reverseRequest will cause the switch to change its position corresponding to the one required by the active route and lock the switch against any other requests.

\[
\text{normalRequest} := \bigvee_{r \in \text{requiringRoutesNormal}} r.\text{reserved}
\]

\[
\text{reverseRequest} := \bigvee_{r \in \text{requiringRoutesReverse}} r.\text{reserved}
\]

The transition unlockContract will open the switch for new commands if both connected tracks are clear.

\[
\text{unlockContract} := \text{Track1.isClear} \land \text{Track2.isClear}
\]

3.4 Route Scheduling

**railML relation** As mentioned before, the railML Interlocking scheme is not finished, i.e., not available, and therefore we have to process simple csv data. This has the same structure as given in the route table in Fig 1b.

**Behavior** A route controls and observes all track side elements necessary for a save train movement. It can be in one of three states (idle, commanded, or occupied). A route is initialized as idle. As previously mentioned, the routes are commanded nondeterministic.
If a route can get active, i.e., is in the state \textit{commanded}, it causes the corresponding tracks to be reserved for this route and the switches to change their position as required. The route will check the state of the elements (\textit{tracksReserved} and \textit{switchesLocked}).

\begin{align*}
\text{tracksReserved} & := \bigwedge_{t \in \text{includedTracks}} t.\text{reserved} \\
\text{switchesLocked} & := \bigwedge_{s \in \text{includedSignals}} s.\text{locked}
\end{align*}

If everything is correct the start signal is changed to proceed.

\begin{align*}
\text{signalProceed} & := \bigwedge_{s \in \text{includedSignals}} s.\text{proceed}
\end{align*}

After that, the route will be occupied. When the train passed the complete route, the route will change its state back to idle.

\begin{align*}
\text{routePassed} & := \bigwedge_{t \in \text{includedTracks}} t.\text{isClear}
\end{align*}

3.5 Safety Specifications

In the following, we want to present the safety properties we provide for our method. Since the defined model is built after a given mechanism, we think it can be easily extended with other safety specifications as the presented ones.

3.5.1 Collision Detection

Two trains collide if they are at the same time at the same place. As described in 3.1 there are two or more track objects for one track element in the track layout to save additional information about the direction of the trains. Therefore, a collision occurs if a track is occupied by more than one train. With this definition, you can detect collisions on switches and head-to-head collisions.

To detect rear-end collisions, tracks have to recognize if a collision can occur in the next step. This is done by observing the reserved and occupied state of the track elements. If a track should get occupied in the next step, \textit{occupiedContract.a} gets true. If it is also already occupied by another train (\textit{track.state} = \textit{occupiedBy.b}) we found a possible collision in the next step.

\begin{align*}
\text{collision}(\text{Track}) & := \\
& (t.\text{occupiedContract.a} \& t.\text{state} = \text{occupiedBy.b}) \\
& | \left(\neg (t.\text{occupiedContract.a} \& t.\text{state} = \text{occupiedBy.b}) \land \right.
\end{align*}

\begin{align*}
& \left. t.\text{occupiedContract.b} \& t.\text{state} = \text{occupiedBy.a} \right)
\end{align*}

3.5.2 Derailment Detection

In this model derailment due to wrong switch positions is checked. The two cases which can happen are i) derailment because a train is moving over a switch while it is changing its position and ii) derailment because a train is passing a switch which is in a wrong position.

The specification will ensure that the position of a switch correlates with the track object which is occupied. So it will check if the switch is in the correct position according to the direction of the train. In the transformation, we derive the correct position for a switch from the route table and encode this within the model. For readability, we refer here to \textit{inRightPos} that evaluate to true if the switch has the required position. As a second aspect, the specification will check if the switch is locked every time a train is on a track linked to this switch.

\begin{align*}
\text{isDerailed} & := \bigvee_{t \in \text{Tracks}} (t.\text{isOccupied} \\
& \land \bigwedge_{s \in \text{switch}(t)} s.\text{inRigthPos} \land s.\text{isLocked})
\end{align*}

3.5.3 Flank Protection Check

Especially on sidings, some parked wagons can start rolling unadvisedly. To prevent accidents between those wagons and trains specific switches which are not on the route of the train can be commanded to a position so that they route the wagons to the other track than used by the train. Nearly the same problem is caused by trains which miss stopping at red signals.

\begin{align*}
\text{Flank protected.} & \quad \text{Flank not protected.} \\
\text{Flank protection available.} & \quad \text{Flank protection not available.}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{flank_protection}
\caption{Verification of flank protection.}
\end{figure}

For checking this safety guideline every time a train is on a switch a backward search is performed. This backward search will start at the not used branch of the switch and will progress hand over hand along the tracks. If it reaches a switch which is in the position to guide wagons away from the train, it will stop (8a). If the switch is in the other position, the search will proceed (8b). If the search reaches a uniting switch, it will continue on both branches (8c). The specification ensures that the backward search will never reach a station or main track.
This formula is also derived automatically from the track layout for each route, i.e., for each route that is active, all adjacent tracks, which are not directly on the route, must be opened. Therefore, for each switch a function \( isFlankProtected(r) \) is defined. It returns true if the switch in in the correct position.

\[
\bigwedge_{s \in \text{Switch}} s.isFlankProtected(r)
\]

4 CASE STUDY BRAINE L’ALLUDE

The validation of the presented modeling methods was executed on a real-world case study of the Belgian train station Braine l’Alleud, taken from (Cappart and Schaus 2016). Braine l’Alleud station consists of four platforms, twelve switches, twelve signals, and 18 track segments. From the given layout, 32 different routes are available.

From this layout, we generated a model with about 100 state machines. This contains representations of the track elements, switches, routes, etc. In the following, we present the verification times of different model checking engines for the defined specifications from sec. 3.5. For providing information about the computation times and capabilities of different verification methods, different tools implementing diverse qualitative model checking techniques were chosen. Further, we checked these specifications on a correct model, i.e., the specifications hold for the system. On the other hand, we injected faulty behavior to provoke erroneous behavior. The failures can be categorized as follows:

- a route does not reserve a needed track
- a route requests a wrong position of a switch
- a route does not request any position of a needed switch

To check the verification time for the erroneous models, we also model checked the specifications and calculated the mean time for each specification, not separating for the found error (i.e., whether a switch or a track was not commanded correctly). We hope that the evaluation of different tools may help users, which are not familiar with model checking, in choosing a proper tool for their purpose.

4.1 Experiments with the Model Checking Tools

To check the modeled interlocking system the model checking tools iimc, nuXmv, and aigbmc were used. The tests were performed on a computer with Intel i7 core (3.2 GHz). The target language of our prototype is the System Analysis and Modeling Language (SAML) (Güdemann and Ortmeier 2010). Using SAML, we can verify the imported qualitative model with several states of the art model checking tools (nuXmv, iimc, UUPAAL), for which the SAML IDE VECS (Verification Environment for Critical Systems) provides implemented connectors. We used this connectors for the experiments with the different model checking tools.

aigbmc This tool is a bounded model checker built on top of the AIGER distribution provided by (Biere 2007). Being a bounded model-checker makes it necessarily incomplete (if the system diameter is unknown), but allows for very efficient counter-example search, as only the base case is checked and the induction step is not encoded.

iimc This tool, described by Hassan et al. in (Hassan et al. 2012), is the evolution of the original IC3 method. It uses different proof engines (BMC, IC3, FAIR), depending on the type of property to verify. The tool is one of the most efficient model-checkers for sequential circuits and allows for multi-threaded verification.

nuXmv This tool is the evolution of NuSMV, described by Cavada et al. (Cavada et al. 2014). It supports infinite state spaces via real-valued variables with an analysis based on the SMT solver MathSAT5 and also PDR-style verification using SMT solvers. This is described by Cimatti and Griggio (Cimatti and Griggio 2012) as IC3 for software verification.

4.2 Evaluation of the Results

In the following, we present the results of the verification experiments. Table 1 presents the verification run time for the different tools and specifications. In this diagram, the BMC-based algorithms are missing since BMC is not capable of proving the correctness of a system rather than falsifying specifications. One most important fact is that all tools that terminated for a given specification also produced correct verification results for both, the correct and the incorrect model. For the verification of the correct model iimc performed best for all specifications. It took between 120s for the collision specification and 608s for the flank protection, which is a quite acceptable computation time. In comparison, the nuXmv model checker had quite more difficulties with about 2400s for the derailment specification (iimc: 120s). Additionally, all tools took the highest computation time between 120s for the collision specification and 608s for the derailment specification (iimc: 120s).

<table>
<thead>
<tr>
<th>Spec</th>
<th>iimcIC3</th>
<th>iimcBMC</th>
<th>nuXmv</th>
<th>AIGBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>collision</td>
<td>120s</td>
<td>—</td>
<td>1419s</td>
<td>—</td>
</tr>
<tr>
<td>collision*</td>
<td>45s</td>
<td>2s</td>
<td>52s</td>
<td>51s</td>
</tr>
<tr>
<td>derailment</td>
<td>120s</td>
<td>—</td>
<td>2402s</td>
<td>—</td>
</tr>
<tr>
<td>derailment*</td>
<td>38s</td>
<td>3s</td>
<td>33s</td>
<td>29s</td>
</tr>
<tr>
<td>flank protect</td>
<td>608s</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>flank protect*</td>
<td>223s</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 1: Verification results of the evaluation. The star * marks the specifications of the defect model.

2https://cse.cs.ovgu.de/vecs
for the flank protection specification, where especially nuXmv was not able to finish its computation within 24 hours.

For the verification of the incorrect model we also used the BMC-based tools. The results show that for medium complex formulae, as the derailment or the flank protection specification, BMC-based algorithms outperformed the inductive ones. However, they were not able to complete their analysis for the flank protection (2s imec BMC compared to 45s imec IC3 for the derailments specification). This behavior is not a surprise. The reason is that the BMC algorithm is very fast for simple specifications and short counterexample paths (10 to 11 steps for derailment and collision). However, the algorithm is not very applicable for complex specifications and, moreover, with high bounds, it was not able to compute the results for the flank protection specification within 24h. This is connected to the algorithm itself since for each step towards the bound, new formulas are added increasing the complexity of the underlying SAT problem. In contrast to that, it is quite fast for the short counterexample since the computation of the invariants is quite complex, without a direct correlation to the number of steps.

Summarizing, BMC has shown up to be a suitable method for finding bugs in a system, especially if the error bound is small. Nevertheless an IC3 implementation is needed for proving the correctness of the system and, moreover, even for the discovery of errors in a deep bound, e.g., the presented flank protection faults. This means imec would be the best tool out of our small collection for the verification of the interlocking systems since it contains both, a fast BMC and a fast IC3 engine.

5 CONCLUSION

In this paper, we presented an approach for reducing the complexity of the formal verification of railroad interlocking systems. Therefore, we presented an approach for automatically generating a formal model from a given track layout and the corresponding route definitions. Moreover, definition rules for basic safety specifications, i.e., derailment, collision avoidance, and flank protection were presented.

To validate our method we generated a model of a real-world case study of the Belgian train station Braine l’Allude. For giving an insight view into available verification algorithms and tools, we verified the specifications with four state-of-the-art verification tools implementing the currently most effective algorithms, IC3 and Bounded Model Checking. In addition to the experiments where the specifications hold, we also examined the behavior of the verification for the error case and injected failures into the model.

The results show, in our point of view, the applicability of the transformation and the computability of the verification. This opens new perspectives for the analysis of interlocking systems since the presented target modeling language SAML supports both, qualitative as well as quantitative verification methods.

REFERENCES


