On the Use of Static Checking in the Verification of Interlocking Systems

Haxthausen, Anne Elisabeth; Østergaard, Peter H.

Published in:
Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)

Link to article, DOI:
10.1007/978-3-319-47169-3_19

Publication date:
2016

Document Version
Peer reviewed version

Citation (APA):
Haxthausen, A. E., & Østergaard, P. H. (2016). On the Use of Static Checking in the Verification of Interlocking Systems. In Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016): Discussion, Dissemination, Applications - Part II (pp. 266-278). Springer. (Lecture Notes in Computer Science, Vol. 9953). DOI: 10.1007/978-3-319-47169-3_19

On the Use of Static Checking in the Verification of Interlocking Systems

Anne E. Haxthausen⋆ and Peter H. Østergaard⋆

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark.
aeha@dtu.dk

Abstract. In the formal methods community, the correctness of interlocking tables are typically verified by model checking. This paper suggests to use a static checker for this purpose and it demonstrates for the RobustRailS verification tool set that the execution time and memory usage of its static checker are much less than of its model checker. Furthermore, the error messages of the static checker are much more informative than the counter examples produced by classical model checkers.

1 Introduction

An *interlocking system* is one of the central components of any railway signalling system. It is responsible for guiding trains safely through a given railway network such that train collisions and derailments are avoided. Therefore interlocking systems have the highest safety integrity level (SIL4) according to the CENELEC 50128 standard [3] for railway applications. The safety verification of interlocking systems represents a considerable challenge and for such SIL4 applications CENELEC 50128 strongly recommends the use of formal methods. Therefore, *formal verification* of interlocking systems is an active research topic investigated by several research groups, e.g. in [23, 17, 1, 9, 10, 21, 2, 13, 11, 5, 6, 12]. An overview of the trends in this research field can be found in [4]. In this paper we will consider how static checking can be used as a part of the formal verification of interlocking system designs.

Conventional development and verification of interlocking systems. Typically for a product line of interlocking systems, each interlocking system consists of (1) a generic part that is the same for all instances of the product line and (2) a part which depends on the network under control and the routes through this and therefore is specific for that system. The generic part is developed and verified once-and-for-all. For an interlocking instance of a product family, first the layout of the network under control is specified and then the specific part is specified by a so-called *interlocking table*[1]. The interlocking table describes the allowed train routes in the network and the specific control rules that the

⋆ The authors' research has been funded by the RobustRailS project granted by Innovation Fund Denmark.

[1] Interlocking tables are also sometimes called *control tables*.

interlocking instance must obey. Later the specific part is developed according to the specification expressed by the interlocking table and then integrated with the generic part. One of the verification tasks in this development is to verify that the interlocking table does not contain errors, while another verification task is to verify that the instantiated system is safe. The first verification task is conventionally manual and informal, while the second task is conventionally done by testing.

Automated, formal verification of interlocking systems. As manual, informal verification is time-consuming and error-prone, and testing first can be done after implementation, automated, formal verification of these tasks in the design phase is an active research topic. Some research groups, e.g. [17, 13, 2, 5, 11], verify the interlocking tables by translating these into interlocking design models and then formally verify by model checking that these models are safe. However, there might be errors in the interlocking tables that can't be found by model checking as they do not lead to safety violations. Also it is a well-known problem that for large networks model checking takes long time and uses much memory (and in worst case fails due to the state-space-explosion problem).

Instead of verifying interlocking tables by means of model checking, we suggest to use a static checker that is able to catch all kinds of data errors in interlocking tables, also those that do not lead to safety violations and therefore can't be found by model checking. The use of a static checker not only has the advantage that it can find more kinds of errors, it is also faster and uses less memory than model checking (as experiments in this paper will show). Time is especially saved, if there are several errors in the interlocking table, as a static checker in contrast to a model checker typically can find all of the errors in one run. In a second step, after having verified the correctness of an interlocking table, we suggest to use a model checker to catch errors in the designed control algorithms of the interlocking model instance that can be derived from the control table. An example of the latter kind of error could be a missing check for the status of conflicting routes in the route allocating protocol. This could lead to a safety violation even the interlocking table were correct. So to detect this kind of error the model checking step is needed. The static checking in the first step is needed as the table can contain errors that do not lead to safety violations and therefore can't be caught by model checking. Hence, the two steps complement each other.

We have previously [7, 8] used this idea for the old relay-based Danish interlocking systems and recently [21] in the RobustRailS[2] project for the interlocking systems of the new Danish ERTMS/ETCS level 2 based signalling systems which are currently being developed. In this paper we will analyse the advantages of using a static checker, and for the RobustRailS tool suite we will make some experiments comparing the execution time and memory usage when using the static checker and the model checker of the RobustRailS tool suite, respectively, to catch errors in interlocking tables. Note that in this comparison we use the

[2] `http://robustrails.man.dtu.dk`

model checker in a way for which it is not intended. We compare with this use of the model checker as this use is the way other research groups usually catch errors in interlocking tables. To our best knowledge such an analysis and comparison has not been made before.

Paper overview. The remainder of the article is organised as follows: First, in Section 2, we introduce the RobustRailS verification tools and basic notions of the railway domain, including the notions of track plans and interlocking tables. Next, in Section 3 and Section 4, we describe the static checker and report on experimental results comparing the execution time and memory usage of the static checker and the model checker for the RobustRailS tool suite. Finally, in Section 5, we make a conclusion.

2 Background

This section gives a brief introduction to (1) the new Danish railway networks, (2) interlocking tables, and the (3) RobustRailS verification method and tools.

2.1 Railway Networks

A railway network in ETCS Level 2 consists of a number of track-side elements like linear sections, points, and marker boards. Figure 1 shows an example layout of a railway network having six linear sections (`b10`,`t10`,`t12`,`t14`,`t20`,`b14`), two points (`t11`,`t13`), and eight marker boards (`mb10`, ..., `mb21`). These terms, as well as their functionality within the railway network, will be explained in more detail in the next paragraphs.

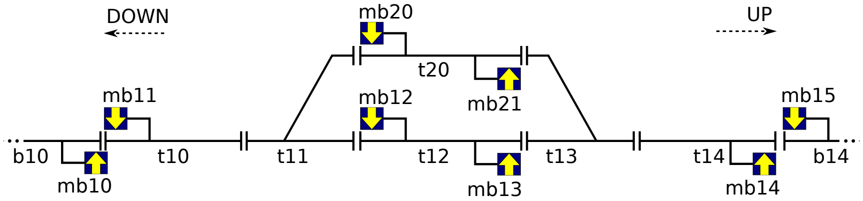


Fig. 1. An example railway network layout.

A *linear section* is a section with up to two neighbours. For example, the linear section `t12` in Figure 1 has `t13` and `t11` as neighbours. A *point* can have up to three neighbours: one at the *stem*, one at the *plus* end, and one at the *minus* end, e.g., point `t11` in Figure 1 has `t10`, `t12`, and `t20` as neighbours at its stem, plus, and minus ends, respectively. The ends of a point are named so that the *stem* and *plus* ends form the straight (main) path, and the *stem* and *minus* ends form the branching (siding) path. A point can be switched between two positions: PLUS and MINUS. When a point is in the PLUS (MINUS) position, its *stem* end is connected to its *plus* (*minus*) end, so traffic can run from its *stem* end to its *plus* (*minus*) end and vice versa. It is not possible for traffic to run from *plus* end to *minus* end and vice versa. Linear sections and points are collectively called *(train detection) sections*, as they are provided with train detection equipments used by the interlocking system to detect the presence of trains on the sections.

Along each linear section, up to two *marker boards* (one for each direction) can be installed. A marker board can only be seen in one direction and is used as reference location (for the start and end of routes) for trains going in that direction. For example, in Figure 1, marker board `mb13` is installed along section `t12` for travel direction up. Contrary to legacy systems, there are no physical signals in ETCS Level 2, but interlocking systems have a *virtual signal* associated with each marker board. Virtual signals play a similar role as physical signals in legacy systems: a virtual signal can be OPEN or CLOSED, respectively, allowing or disallowing traffic to pass the associated marker board. However, trains (more precisely train drivers) do not see the virtual signals, as opposed to physical signals. Instead, the aspect of virtual signals (OPEN or CLOSED) are communicated to the onboard computer in the train via a radio network. For simplicity, the terms *virtual signals*, *signals*, and *marker boards* are used interchangeably throughout this paper.

2.2 Interlocking Tables

An interlocking system monitors constantly the status of track-side elements, and sets them to appropriate states in order to allow trains travelling safely through the railway network under control. A common approach of interlocking systems is to use the concept of train routes, where a *(train) route* is a path from a *source (entry)* signal to a *destination (exit)* signal in the given railway network. The idea is that trains should travel along predefined routes in the network. Before a train can enter a route, the route must first be *set*, i.e. resources such as sections, points, and signals must be allocated (set to appropriate states) for the route and then *locked* exclusively for that train. For more details on the interlocking principles for the new Danish systems, see [20–22, 19].

An *interlocking table* specifies routes in the railway network under control and the conditions for setting these routes. The specification of a route r and conditions for setting r include the following information:

- $id(r)$ – the route's unique identifier,
- $src(r)$ – the source/entry signal of r,
- $dst(r)$ – the destination/exit signal of r,
- $path(r)$ – the list of sections constituting r's path from $src(r)$ to $dst(r)$,
- $overlap(r)$ – a list of the sections in r's overlap, i.e., the buffer space after $dst(r)$ that would be used in case trains overshoot the route's path,
- $points(r)$ – a map from points[3] used by r to their required positions,

[3] These include points in the path and overlap, and points used for flank and front protection. For detail about flank and front protection, see [16].

id	src	dst	path	points	signals	conflicts
1a	mb10	mb13	t10;t11;t12	t11:p;t13:m	mb11;mb12;mb20	1b;2a;2b;3;4;5a;5b;6b;7
1b	mb10	mb13	t10;t11;t12	t11:p	mb11;mb12;mb15;mb20;mb21	1a;2a;2b;3;5a;5b;6a;6b;7;8
2a	mb10	mb21	t10;t11;t20	t11:m;t13:p	mb11;mb12;mb20	1a;1b;2b;3;5b;6a;6b;7;8
2b	mb10	mb21	t10;t11;t20	t11:m	mb11;mb12;mb13;mb15;mb20	1a;1b;2a;3;4;5a;5b;6a;6b;7
3	mb12	mb11	t11;t10	t11:p	mb10;mb20	1a;1b;2a;2b;5a;6b;7
4	mb13	mb14	t13;t14	t13:p	mb15;mb21	1a;2b;5a;5b;6a;6b;8
5a	mb15	mb12	t14;t13;t12	t11:m;t13:p	mb13;mb14;mb21	1a;1b;2b;3;4;5b;6a;6b;8
5b	mb15	mb12	t14;t13;t12	t13:p	mb10;mb13;mb14;mb20;mb21	1a;1b;2a;2b;4;5a;6a;6b;7;8
6a	mb15	mb20	t14;t13;t20	t11:p;t13:m	mb13;mb14;mb21	1b;2a;2b;4;5a;5b;6b;7;8
6b	mb15	mb20	t14;t13;t20	t13:m	mb10;mb12;mb13;mb14;mb21	1a;1b;2a;2b;3;4;5a;5b;6a;8
7	mb20	mb11	t11;t10	t11:m	mb10;mb12	1a;1b;2a;2b;3;5b;6a
8	mb21	mb14	t13;t14	t13:m	mb13;mb15	1b;2a;4;5a;5b;6a;6b

Table 1. Interlocking table generated for the network layout in Figure 1. The overlap column is omitted as it is empty for all of the routes. (`p` means PLUS, `m` means MINUS.)

- *signals*(r) – a set of protecting signals used for flank or front protection [16] for the route, and
- *conflicts*(r) – a set of conflicting routes which must not be set while r is set.

Table 1 shows an example of an interlocking table for the network shown in Figure 1. Each row of the table corresponds to a route specification. The column names indicate the information of the route specifications that these columns contain. As can be seen, one of the routes has id `1a`, goes from `mb10` to `mb13` via three sections `t10`, `t11` and `t12` on its path, and has no overlap. It requires point `t11` (on its path) to be in PLUS position, and point `t13` (outside its path) to be in MINUS position (as a protecting point). The route has `mb11`, `mb12` and `mb20` as protecting signals, and it is in conflict with routes `1b`, `2a`, `2b`, `3`, `4`, `5a`, `5b`, `6b`, and `7`.

2.3 The RobustRailS Verification Method and Tool Set

This section describes shortly the RobustRailS verification method and tool set. For more information, see [20–22, 19].

The tools are centred around a domain-specific language (DSL) for representations of network diagrams and interlocking tables as described in the preceding subsections. The tools comprise among others:

- *A static checker* for checking that a DSL specification follows certain general wellformedness rules.
- The bounded *model checker* of RT-Tester [14, 18] which is set up such that it can make a k-induction proof.
- *Generators* which from a DSL specification produce input to the model checker: (1) a formal, behavioural model of interlocking system and its environment and (2) required safety properties expressed as formulae in the temporal logic LTL.

There are additional tools supporting automated, model-based testing of implemented interlocking systems, see [19].

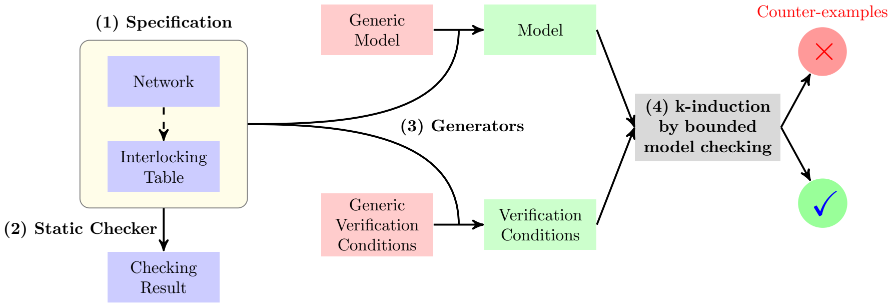


Fig. 2. The RobustRailS verification process.

The tools can be used to verify the design of an interlocking system in the following number of steps, as illustrated in Fig. 2:

1. Write a DSL specification of the interlocking system:
   (a) first the network layout,
   (b) and then the interlocking table (this is either done manually or generated automatically from the network layout)
2. Validate the specification using the static checker.
3. Apply the generators to generate input to a model checker.
4. Apply the model checker to that input to investigate whether the model satisfies the required safety properties.

The static checking in step (2) is intended to catch errors in the network layout and interlocking table, while the model checking in step (4) is intended to catch safety violations in the control algorithm of the instantiated model.

3 Static Checker

This section describes the static checker of the RobustRailS tool set.

The static checker takes as input a network layout and an associated interlocking table and checks whether these are well-formed. In case there are errors, it suggests what might be wrong and in some cases also how this can be fixed. It checks for instance that the network layout represents a legal railway network of track elements and that the interlocking table satisfies the following conditions:

Elements Exist It refers only to existing track elements in the network layout.
Path The path of a route is a connected path in the network layout.
Overlap The overlap of a route must be a connected path that continues right after the last section of the route path itself.
Entry/Exit The entry/exit signal of a route must be at the start/end of the path of the route and be visible in the direction of the route.
Elementary The route must be elementary, i.e. between the entry signal and exit signal of a route, there must not be any signal visible in the direction of the route.

Points Points in the path and overlap of a route must appear in the points field of the route and the required position of each of these points must fit the path of the route (to avoid derailments).

Front/Flank Protection For each route a sufficient front and flank protection must be given by (1) the signals listed in the signals field and/or (2) required point settings (in the points field) for points outside the route.

Route Conflicts Routes that are in conflict[4] with a route must be listed in the conflicts field of the route.

The checker has been formally specified in the RAISE Specification Language, RSL [15], as described in [20, 19], and implemented in C++.

3.1 Examples of error messages from the static checker

Below we give examples of some illegal interlocking tables for the network layout given in Fig. 1 and show the error messages that the static checker gives for these tables. In each case it is explained how the illegal interlocking table is obtained from Table 1 by modifying some of the fields for route `1a`.

If a table contains several errors, the static checker will provide error messages for each of these.

*Example breaking the **Path** condition:* Remove `t10` in the path field. For this the static checker provides the error message:

```
In route 1a, two consecutive segments, b10 and t11, are not
connected.
```

*Example breaking the **Point** condition:* Remove `t11` in the points field. For this the static checker provides the error message:

```
For route 1a, point t11 is not given a point position.
```

*Example breaking the **Point** condition:* Require point `t11` to be in a wrong position (`m` rather than `p`). Note that this change means that route `1a` now also becomes in conflict with route `6a`, as these two routes require `t11` to be in different positions, but in the interlocking table we have not listed them as being in conflict. Therefore, the static checker provides the error message:

```
For route 1a, point t11 is set to MINUS,
but it should have been set to PLUS.

Routes 1a and 6a are in conflict, but are not listed as being in
conflict. Reasons to be in conflict:
- Shared point required in different positions: t11.
```

[4] This essentially means that they use the same track elements. For a complete definition, see [19].

Example breaking the ***Route Conflicts*** *condition:* Remove route 7 in the conflicts field (and remove 1a from the conflicts field of route 7). For this the static checker provides the error message:

```
Routes 1a and 7 are in conflict, but not listed as being in
conflict. Reasons to be in conflict:
- Non-concatenated routes with shared elements: {t10, t11}.
- Entry signal of one route used as protecting signal for the other
  route.
```

Example breaking the ***Flank Protection*** *condition:* Remove protecting point t13 in the points field. Note with this change, routes 1a and 4 are not anymore in conflict. For this the static checker provides the error message:

```
Routes 1a and 4 are listed as being in conflict, but they are not.

Improper protection of section t12 in route 1a.
Possible protections:
--------------------------
Signals: {}
Points: {t13:m}
--------------------------
Signals: {mb15, mb21}
Points: {}
```

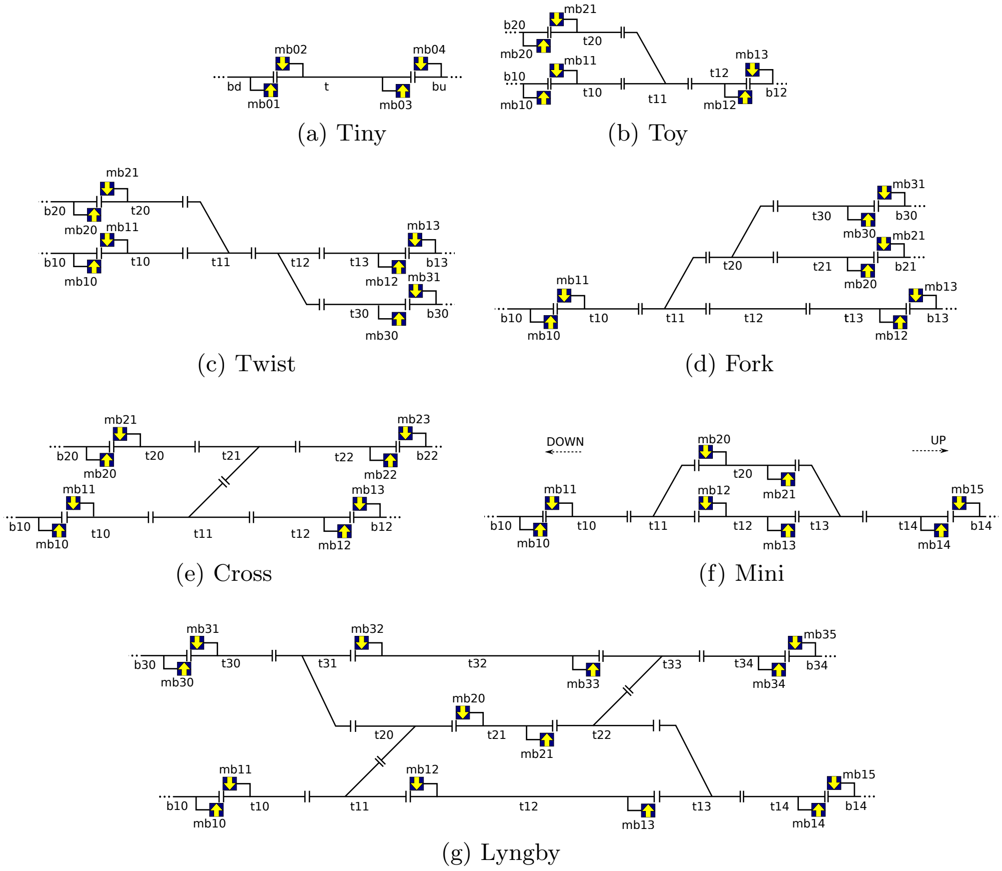
As it can be seen, two alternative ways of obtaining a flank protection for section t12 are suggested. The first one corresponds to re-introducing what was removed, while the second one corresponds to the protection in route 1b.

4 Experiments

This section compares for a selection of railway networks the execution time and memory usage of the static checking and the model checking of these.

4.1 Selection of networks

Ten cases of networks from [19] have been selected. The layout of the seven smallest cases are shown in Figure 3. These seven networks are inspired by the typical examples used in other studies about formal verification of railway interlocking systems. The three last networks are some real examples from Denmark: Gadstrup-Havdrup and Køge are extracted from the Early Deployment Line (EDL) in the Danish Signalling Programme. The EDL is the first regional line in Denmark to be commissioned in the Danish Signalling Programme. The line goes from Roskilde station to Næstved station and is over 55 kilometres long. It includes eight stations ranging from simple stations similar to the one shown in Figure 1, to complex stations such as Køge. The



(a) Tiny (b) Toy

(c) Twist (d) Fork

(e) Cross (f) Mini

(g) Lyngby

Fig. 3. The seven smallest railway network cases.

	Linears	Points	Signals	Routes	States
Tiny	3	0	4	2	10^{11}
Toy	6	1	6	4	10^{26}
Twist	8	2	8	8	10^{39}
Fork	9	2	8	6	10^{40}
Cross	8	2	8	10	10^{41}
Mini	6	2	8	12	10^{37}
Lyngby	11	6	14	24	10^{79}
Gadstrup-Havdrup	21	5	24	33	10^{113}
Køge	57	23	60	73	10^{332}
EDL	110	39	126	179	10^{651}

Table 2. Metrics of network cases.

network descriptions (in XML representation) and the corresponding generated properties and model instances for the first seven cases can be found at `http://www.imm.dtu.dk/~aeha/RobustRailS/data/casestudy`.

Table 2 lists the following metrics for each of the selected networks: The number of linear sections, points, and signals, the number of routes in the interlocking table that can be generated from the network, and the number of states in the model that can be generated from the network and the interlocking table.

4.2 Experiments with correct networks and interlocking tables

For each of the networks we first used the interlocking table generator to generate an interlocking table for that network. Then for each case we used

1. the static checker to verify correctness conditions for the network and interlocking table and
2. the generator tools to generate a model and safety conditions, whereupon we used the model checker to check the model against the safety conditions.

In all cases the static check and the model check confirmed that there were no static errors in the network layout or interlocking table and that the instantiated model was safe, respectively. All experiments have been performed on a machine with Intel(R) Xeon(R) CPU E5-2667 @ 2.90GHz, 64GB RAM, CentOS 6.6, Linux 2.6.32-504.8.1.el6.x86_64 kernel.

	Static Checker		Model Checker		Time Ratio
	Time	Memory	Time	Memory	
Tiny	0.20	–	0.74	18.4	3.7
Toy	0.52	–	2.78	86.3	5.4
Twist	0.24	–	9.76	170.4	41
Fork	0.22	–	8.80	168.8	40
Cross	0.28	–	14.48	191.8	52
Mini	0.26	–	17.56	197.4	68
Lyngby	0.30	–	254.3	868.1	848
Gadstrup-Havdrup	0.36	–	230.1	1146	639
Køge	0.34	–	5528.5	8471.6	16260
EDL	0.43	–	19358.9	23389.4	45020

Table 3. Execution times and memory usage.

Table 3 shows for each of the network cases, the approximate real execution time (in seconds) and memory usage (in MB) for the static checking and the model checking (incl. model generation), respectively. Furthermore, the last column shows the ratio between the execution time of the model check and the execution time of the static check. As it can be seen the static checking is much faster than the model checking. For the smallest network it is a factor 3.7 faster, and then it increases up to a factor 45020 for the largest network. The static checker ran too quick for the profiler tool (*runlim*) to measure the memory usage.

4.3 Experiments with illegal interlocking tables

We also tried to inject some errors in the interlocking tables. Also in this case the static checker was much faster to catch errors than the model checker. For instance, if we for route 1a in Table 1 require `t11` to be `m` instead of `p` in the points field, then the static checker detects this in 0.14 seconds, while it takes the model checker 20.54 seconds. I.e. the static checker is a factor 146.7 faster.

In some cases the errors are even not caught by the model checker, as the wrong data does not lead to a safety violation in the instantiated model. The reason for this is that the interlocking system (and thereby also the model) contains redundant checks to make the system more fault tolerant. This fact also indicates the advantage of using a static checker.

Furthermore, the error messages provided by the static checker are much more informative. They explain exactly what the problem is and in some cases suggest how to fix it. In contrast to that, when the model checker gives a counter example, this has to be analysed to find out what the problem is: first it has to be determined whether the unsafe situation is due an error in the interlocking table or in the model.

5 Conclusion

In the formal methods community, the correctness of interlocking tables are typically verified by model checking. While this is a good method, it suffers from the state space explosion problem for larger networks. This paper has suggested to use a static checker to verify the correctness of interlocking tables. Experiments using the RobustRailS interlocking verification tool set showed for a selection of railway networks with associated interlocking tables that the execution time and memory usage of verifying the interlocking tables using the static checker was much less than of using the model checker[5]. Furthermore, the error messages of the static checker are more informative and do not need an analysis to find out what the error is, as it is the case of the counter examples of the model checker. The static checker can also in contrast to the model checker catch several errors in the same execution. So our conclusion is that for the checking of interlocking tables it is worth to provide such a user-friendly static checker.

Acknowledgements. The authors would like to express their gratitude to (1) Jan Peleska and Linh Hong Vu for the excellent contribution to the development of the RobustRailS interlocking verification method and tool set (including the static checker discussed in this paper) and for an always very enjoyable collaboration, (2) Ross Edwin Gammon and Nikhil Mohan Pande from Banedanmark and

[5] It should here be noted that using this model checker in a second step for verifying the safety of the model of the instantiated interlocking system has actually turned out to be very efficient. For instance, it succeeded to verify the EDL line, where other model checkers failed within some given resources, cf.[22].

Jan Bertelsen from Thales for helping with their expertise about Danish interlocking systems, and (3) Uwe Schulze and Florian Lapschies from the University of Bremen for their help with the implementation in the RT-Tester toolchain.

References

1. M. Banci, A. Fantechi, and S. Gnesi. Some Experiences on Formal Specification of Railway Interlocking Systems Using Statecharts. 2005.
2. Y. Cao, T. Xu, T. Tang, H. Wang, and L. Zhao. Automatic Generation and Verification of Interlocking Tables Based on Domain Specific Language for Computer Based Interlocking Systems (DSL-CBI). In *Proceedings of the IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011)*, pages 511 – 515. IEEE, 2011.
3. C. European Committee for Electrotechnical Standardization. *EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems.* 2011.
4. A. Fantechi. Twenty-Five Years of Formal Methods and Railways: What Next? In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.
5. A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model Checking Interlocking Control Tables. In E. Schnieder and G. Tarnai, editors, *FORMS/FORMAT 2010 – Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 107–115. Springer, 2010.
6. H. H. Hansen, J. Ketema, B. Luttik, M. R. Mousavi, J. van de Pol, and O. M. dos Santos. Automated Verification of Executable UML Models. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 225–250. Springer, 2010.
7. A. E. Haxthausen. Towards a Framework for Modelling and Verification of Relay Interlocking Systems. In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, number 6662 in Lecture Notes in Computer Science, pages 176–192. Springer, 2011.
8. A. E. Haxthausen. Automated Generation of Formal Safety Conditions from Railway Interlocking Tables. *International Journal on Software Tools for Technology Transfer (STTT), Special Issue on Formal Methods for Railway Control Systems*, 16(6):713–726, 2014.
9. A. E. Haxthausen, M. L. Bliguet, and A. A. Kjær. Modelling and Verification of Relay Interlocking Systems. In C. Choppy and O. Sokolsky, editors, *Foundations of Computer Software, Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2010.
10. A. E. Haxthausen, J. Peleska, and S. Kinder. A Formal Approach for the Construction and Verification of Railway Control Systems. *Formal Aspects of Computing*, 23(2):191–219, 2011.
11. P. James, F. Möller, H. N. Nguyen, M. Roggenbach, S. Schneider, H. Treharne, M. Trumble, and D. Williams. Verification of Scheme Plans Using CSP||B. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2014.

12. C. Limbrée, Q. Pecheur, C. Moller, and S. Tonetta. Verification of railway interlocking - Compositional approach with OCRA. In T. Lecomte, R. Pinger, and A. Romanovsky, editors, *Reliability, Safety and Security of Railway Systems - RSSR 2016*, volume 9707 of *Lecture Notes in Computer Science*. Springer, 2016.
13. A. Mirabadi and M. B. Yazdi. Automatic Generation and Verification of Railway Interlocking Control Tables Using FSM and NuSMV. *Transportation Problems*, pages 103–110, 2009.
14. J. Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In A. K. Petrenko and H. Schlingloff, editors, Proceedings 8th Workshop on Model-Based Testing, Rome, Italy, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association, 2013.
15. The RAISE Language Group: Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., 1992.
16. G. Theeg, S. V. Vlasenko, and E. Anders. *Railway Signalling & Interlocking: International Compendium*. Eurailpress, 2009.
17. D. Tombs, N. Robinson, and G. Nikandros. Signalling Control Table Generation and Verification. In *CORE 2002: Cost Efficient Railways through Engineering*, page 415. Railway Technical Society of Australasia/Rail Track Association of Australia, 2002.
18. Verified Systems International GmbH. *RT-Tester Model-Based Test Case and Test Data Generator - RTT-MBT - User Manual*, 2013. Available on request from `http://www.verified.de`.
19. L. H. Vu. *Formal Development and Verification of Railway Control Systems - In the context of ERTMS/ETCS Level 2*. PhD thesis, Technical University of Denmark, DTU Compute, 2015.
20. L. H. Vu, A. E. Haxthausen, and J. Peleska. A Domain-Specific Language for Railway Interlocking Systems. In E. Schnieder and G. Tarnai, editors, *FORMS/FORMAT 2014 - 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 200–209. Institute for Traffic Safety and Automation Engineering, Technische Universität Braunschweig, 2014.
21. L. H. Vu, A. E. Haxthausen, and J. Peleska. Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release. In C. Artho and P. C. Ölveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 476 of *Communications in Computer and Information Science*, pages 223–238. Springer International Publishing, 2015.
22. L. H. Vu, A. E. Haxthausen, and J. Peleska. Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release. *Science of Computer Programming*, 2016. http://dx.doi.org/10.1016/j.scico.2016.05.010.
23. K. Winter, W. Johnston, P. Robinson, P. Strooper, and L. van den Berg. Tool Support for Checking Railway Interlocking Designs. In *Proceedings of the 10th Australian workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 101–107, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.