# A dedicated algorithm for verification of interlocking systems

Quentin Cappart and Pierre Schaus*

Université catholique de Louvain, Louvain-La-Neuve, Belgium
{quentin.cappart|pierre.schaus}@uclouvain.be

**Abstract.** A railway interlocking is the system ensuring a safe train traffic inside a station by monitoring and controlling signalling components such as the signals or the points. Modern interlockings are controlled by a generic software that uses data, called application data, reflecting the layout of the station under control and defining which actions the interlocking can perform. The safety of the train traffic relies thereby on application data correctness, errors inside them can lead to unexpected events, such as collisions or derailments. However, the application data are nowadays prepared by automatic tools that do not guarantee a sufficient level of safety. Furthermore, their verification is a time consuming task and error prone as it is mostly performed by human testers. Given the high level of safety required by such a system, verification of application data is a critical concern. For such reasons, automatising and improving the verification process of application data is an active field of research. Most of this research is based on model checking, which performs an exhaustive verification of the system but which suffers from scalability issues. However, even if such an approach requires knowledge of the interlocking behaviour for modelling the system, it does not take advantage of it for the verification itself. In this paper, we propose to use our knowledge of the system in order to design a scalable verification algorithm. Concretely, we develop a polynomial algorithm that can detect all the possible safety issues provided that an assumption of monotonicity hold. We finally apply it on a realistic medium size station of the Belgian railway network.

## 1 Introduction

In the railway domain, an interlocking is the subsystem that is responsible for ensuring a safe and fluid train traffic by controlling active track components of a station. Among these components, there are the signals, defining when trains can move, and the points, that guide trains from track to track. Modern interlockings, like Solid State Interlocking [1], are computerised systems composed of a generic software taking data, called application data, as input. They describe

* This research is financed by the Walloon Region as part of the Logistics in Wallonia competitiveness pole.

the actions that the interlocking must perform [2]. The main requirement to consider when designing an interlocking is the safety. A correct interlocking must never allow critical situations such as derailments or collisions. To this purpose, an interlocking must satisfy the highest safety integrity level as stated by Standard EN 50128 of CENELEC [3]. Although the generic software is developed in accordance with these requirements, the reliability of an interlocking is also dependant of the correctness of its application data which are particular to each station. However, preparation of application data is still nowadays done by tools that do not guarantee the required level of safety. Furthermore, the verification of their correctness, as well as their validation, is mainly done manually through a physic simulator that reproduces the behaviour of the interlocking on real infrastructures. In addition to the high cost of this process, it is also error prone because there is no guarantee that all the situations that could end-up in a safety issue have been tested by the simulator.

To overcome this lack, research has been carried out in order to improve this verification process [4–7]. Most of it is based on model checking [8]. The goal is to perform an exhaustive verification of the system. It is done in three steps. First, the application data and the station layout are translated into a model reflecting the interlocking behaviour. Secondly, the requirements that the interlocking must ensure in order to prevent any safety issue are formalised. Finally, the model checker verifies that no reachable state of the model violates the safety requirements. The main advantage of this method is its exhaustiveness: if a requirement is not satisfied, the model checker will always detect it. However, this method suffers from the state space explosion problem. The number of reachable states exponentially grows as the size of the model grows and the model checker algorithm might not return a result within a reasonable time in practice. Different methods to limit it have been proposed. Winter et al. [9] suggest to keep the model as simple as possible by abstracting some parameters, such as the trains speed or length. Besides, improvements can also be done on the model checking algorithms. Different studies propose to use symbolic model checking instead of classical approaches [6, 7]. Variable ordering can also be considered in order to speed up the verification [10]. Cappart et al. [11] propose to limit the verification to a set of likely scenarios through a discrete event simulation. Furthermore, Limbree et al. [12] propose a compositional approach and use modern model checking algorithms, such as IC3 or k-liveness for the verification. However, despite the good performances obtained, their method still requires manual work for modelling each station individually and defining their decomposition through contracts.

All of these improvements are generic and although they can be applied for any model checking application, they do not take advantage of the intrinsic specificities of the considered system. In this paper, we propose to use our knowledge of the railway field in order to design an efficient dedicated verification algorithm. The contributions of this paper are as follows:

- An extension of the model presented by Cappart et al. [11]. Concretely, we add the bidirectional locking functionality [2] that prevents head to head collisions on platforms. We also add the differentiation between a route command and a route activation.
- The introduction of a polynomial algorithm verifying that the interlocking will never cause derailments or collisions provided that an assumption of monotonicity hold. It also verifies that each train will reach its correct destination. Furthermore, its performances are also analysed through several experimentations done on three instances.

This paper is structured around a typical medium sized Belgian station (the same as [11]). The next section describes the interlocking components, explains how it works, and illustrates its behaviour on the case study. Section 3 presents the verification algorithm and states under which assumptions it can be used. Performances are finally discussed in Section 4.

## 2 Interlocking principles

The role of an interlocking is to ensure a safe train traffic inside a station. This section explains how it is done in practice for the Belgian interlockings and illustrates the process on a case study, Braine l'Alleud Station. A representation of its track layout with its related components is shown on Figure 1.

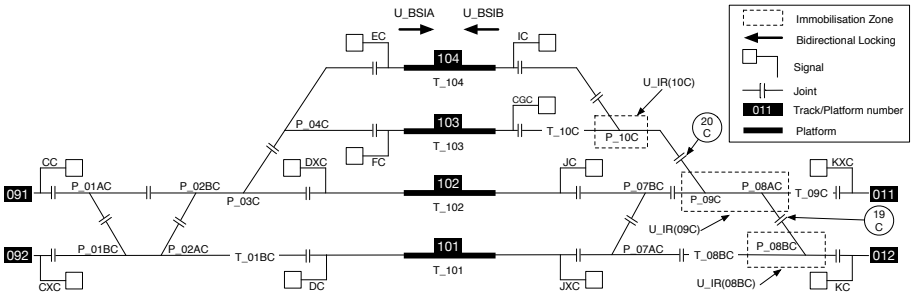


**Fig. 1.** Layout of Braine l'Alleud Station.

This figure recaps all the component types that are used in our model. Firstly, there are the physical components of the track layout:

- The **tracks** (e.g Track 101) are the railway structures where trains can move. A track can be a **platform** if the train can stop on it to pick up passengers.
- The **track segments** (e.g T_01BC) are the portions of tracks where a train can be detected. They are delimited by the **joints**.
- The **points** (e.g. P_01AC) are the movable devices that allow trains to move from one track to another. According to Belgian convention, they can be in a normal position (left) or in a reverse position (right).

- The **signals** (e.g. CXC) are the devices used to control the train traffic. They are set on a proceed state (green) if a train can safely move into the station or in a stop state (red) otherwise.

Braine l'Alleud Station is composed of 4 tracks, 17 track segments, 4 platforms, 12 points and 12 signals. The physical components are controlled and monitored by the interlocking. For instance, the system can detect that a train is waiting on Track segment T_01AC in front of Signal CC and then puts this signal to a proceed state if this action will not cause any safety issue. Generally speaking, the interlocking must known which actions can be done and under which conditions. Such information can be defined in different ways according to the type of interlocking considered. Since 1992, Belgian railway stations have used SSI format [1] for their interlockings. Such interlockings use a route based paradigm. A **route** is the path that a train is supposed to follow inside a station. It is named according to its origin and its destination place. Signals are often used as a reference for the origins whereas tracks or platforms are used for destinations. For instance, Route R_CXC_101 starts from Signal CXC and ends on Platform 101. When a train is approaching to a station, a signalman performs a route request to the interlocking in order to ask if the route can be commanded. It is a **route command**. If this request is fulfilled, all the requested components are locked but the train cannot use the route yet because the start signal is still on a stop state. The start signal goes to a proceed state only after the activation of the route. **Route activations** are periodically tried by the interlocking after that the route has been commanded. Once the route activation has been accepted, the train can finally use its route. The interlocking handles such requests and accepts or rejects it according to the station state. To manage the requests, logical components are used:

- The **subroutes** are the contiguous segments that the trains must follow inside a route. When a route is commanded for a train, a set of subroutes is locked. When not requested, subroutes are in a free state. They are defined by this syntax: U_origin_dest. For instance, U_19C_20C is the subroute from Joint 19C to Joint 20C.
- The **immobilisation zones** are the variables materialising the immobilisation of a set of points. When they are locked, their attached points cannot be moved. They are represented in the application data by the name U_IR.
- The **bidirectional locking** is the mechanism used to prevent head to head collisions on platforms. Each bidirectional locking consists of two variables (U_BSIA and U_BSIB) which can prevent the activation of a route coming from the left or the right of the platform. For instance, when U_BSIA(104) is locked, no route going to the Platform 104 from the right can be activated.

There are 32 possible routes in Braine l'Alleud. To manage it, 48 subroutes, 10 immobilisation zones and 4 bidirectional locking mechanisms are used. With both the physical and logical components, a route based interlocking controls the train traffic by monitoring the station, setting routes, activating them, locking

components and releasing them. To illustrate how it works, let us consider the scenario where a train is coming from Track 012 and has to go to Platform 103:

- Firstly, when the train is waiting at Signal KC, the interlocking verifies whether the request for Route R_KC_103 can be granted. Listing 1.1 presents the request according to the application data of Braine l'Alleud.

```
1  *Q_R(KC_103)
2     if    R_KC_103 xs, // xs: unset
3           P_08BC cfr, P_08AC cfr, P_09C cfr, P_10C cfn,
4           U_IR(08BC) f, U_IR(09C) f, U_IR(10C) // f: free
5     then  R_KC_103 s // s: set
6           P_08BC cr, P_08AC cr, P_09C cr, P_10C cn,
7           U_IR(08BC) l, U_IR(09C) l, U_IR(10C) l,
8           U_KC_19C l, U_19C_20C l, U_20C_CGC l // l: locked
```

**Listing 1.1.** Request for commanding Route R_KC_103.

The request is accepted only if Route R_KC_103 is not already set (line 2), if some points are free to be commanded to the reverse (cfr) or normal (cfn) position (line 3) and if some immobilisation zones are not locked (line 4). If all the conditions are satisfied, R_KC_103 is set (line 5), the points are controlled to the reverse (cr) or normal (cn) position (line 6) and some components as the immobilisation zones (line 7) or subroutes (line 8) are locked. At this step, Route R_KC_103 is set, or commanded, but not yet activated. Its start signal is still on a stop state and the train can thereby not enter in the station yet.

- Before moving a point, the interlocking must verify that this action can safely be executed. Listing 1.2 illustrates such conditions for Point P_08AC.

```
1  *P_08ACN U_IR(09C) f // condition for normal (N) position
2  *P_08ACR U_IR(09C) f // condition for reverse (R) position
```

**Listing 1.2.** Conditions allowing Point P_08AC to move.

- Directly after the acceptance of the request of Listing 1.1, the interlocking checks if a bidirectional locking must be used in order to prevent routes going to Platform 103 from the left to be activated. It is shown on Listing 1.3.

```
1  if U_BSIA(103) f then U_BSIB(103) l
```

**Listing 1.3.** Request for setting the bidirectional locking of Platform 103.

- Once R_KC_103 has been commanded, the interlocking checks if it can safely activate the route and so gives the train an authority to move.

```
1  *R_KC_103
2     if    P_08BC cdr, P_08AC cdr, P_09C cdr, P_10C cdn,
3           U_IR(08BC) l, U_IR(09C) l, U_IR(10C) l,
4           T_08BC c, T_09C c, T_10C c, T_103 c, // c: clear
5           U_BSIA(103) f
6     then  U_BSIB(103) l, KC proceed
```

**Listing 1.4.** Request for activating Route R_KC_103.

Listing 1.4 states that R_KC_103 can be activated only if the points are commanded and detected in the requested position (cdn and cdr on line 2), if the immobilisation zones are locked (line 3), if there is no train on some track segments (line 4) and if the bidirectional locking for trains coming from right to Platform 103 is free (line 5). The route activation results on locking the paired bidirectional locking and on setting Signal KC on a proceed state (line 6). At this step, the train can finally move into the station.

- When they are not used, locked components can be released. It is done according to the progress of the train on its route. After each train movement, the interlocking checks if a releasing event can be triggered. Listing 1.5 states the conditions for releasing Subroute U_20C_CGC. If all the conditions are fulfilled, the requested components are thoroughly released.

```
1  U_20C_CGC f if U_KXC_20C f, U_19C_20C f, T_10C c
```

**Listing 1.5.** Conditions for releasing Subroute U_20C_CGC.

This process briefly describes the life cycle of a route and how it is managed by the interlocking. To be more precise, application data also contain other information but it is either not related to the safety or abstracted in our model. Cappart et al. [11] designed a model aiming to reproduce the interlocking behaviour through a discrete event simulation. However they did not consider the bidirectional locking conditions and the differentiation between a route command and a route activation. In this paper, we enrich their model by adding these functionalities. Errors in application data can lead to disastrous situations. For instance, if the bidirectional locking is not properly checked before activating route R_KC_103 (line 5 missing from Listing 1.4), two routes going to the same platform from a different side can be activated together which will potentially cause a head to head collision. There is thereby a real need of efficient and reliable methods to verify the application data correctness.

## 3 Verification algorithm

This section describes the method that we have designed to verify that an interlocking will never cause safety issue in a station. However, we need to define first what is exactly a safety issue and how it can be detected. Different authors [5, 13, 14] identified two types of safety issues: collisions and derailments. According to Busard et al. [13], there are three requirements that must hold in order to avoid safety issues. Beyond the safety, a correct interlocking must also ensure that trains will always reach their destination. We have then four requirements:

(1) A same track segment cannot have two trains or more on it at the same time. Otherwise, a collision will occur.

(2) A point cannot move if there is a train on it in order to avoid derailments.
(3) A point must always be set on a position allowing trains to continue their path in order to avoid derailments.
(4) Each train following a route must reach the destination stated by the route.

Much research has been carried out in order to verify automatically if an interlocking always satisfies these properties. However, current methods present some shortcomings. Model checking approaches suffer from the state space explosion problem and the discrete event simulation [11] does not provide enough guarantees that all the errors leading to safety issues will be detected. The approach described in this paper tackles the problem with a different perspective. Instead of limiting our knowledge of the system only for its modelling, we propose to use it for designing the verification algorithm. Specificities of the system can be used to identify what are the scenarios that can lead to safety issues and to distinguish them from others that are either redundant or that never happen in practice. The state space is then pruned and the verification is more efficient. This approach is related to model checking. Indeed, an automatic and exhaustive verification is still performed, but now this verification is limited to a limited state space that increases polynomially in function of the number of routes and track segments. The rest of this section describes our algorithm and states the assumption under which it can be used.

*Initialisation* This paragraph presents the variables and the conventions used in our dedicated algorithm. For a station $S$, we define $ROUTES$ as the set of all routes, $TRACK\_SEGMENTS$ as the set of all track segments, $POINTS$ as the set of all points and $COMPONENTS$ as the set of all physical components in $S$. The algorithm returns `True` if $S$ satisfies the requirements and `False` otherwise. For all routes $r$, we define $r.origin$ as the origin of $r$, $r.destination$ as its destination, $r.isCommanded$ and $r.isActivated$ as boolean values defining if $r$ is commanded and activated. We also define $t.position$ as the current position of a train $t$, $p.state$ as the state (normal or reverse) of a point $p$ and $c.isLocked$ as a boolean value defining whether a component $c$ is locked.

*No conflictual pair of routes* The idea behind this algorithm is to verify that no issue occurs in any situation, and for that, only pairs of routes are considered. The correctness of this algorithm is then based on the assumption that testing only pairs of routes is sufficient for detecting all the issues. It is related to the **monotonicity** of the application data.

**Proposition 1.** *The application data are monotonic. If a route cannot be commanded given a particular station state, it will not be able to be commanded for a more constrained station state. The same rule must also apply for the components releasing.*

*Proof.* In other words, if a route $r_1$ cannot be commanded when a route $r_2$ is commanded, it cannot be commanded if $r_2$ and a third route $r_3$ are commanded

together. Such a scenario can only occur if conditions for route commands (Listing 1.1) require components to be locked instead of being free. It is because the station becomes more constrained each time a component is locked for a route. In some cases, the application data are not monotonic. This situation happens when the itinerary of a train is not only determined by a single route but by a sequence of $n$ routes $[r_1, \ldots, r_n]$. In this case, a route $r_i$ with $i \in ]1, n]$ can only be set if $r_{i-1}$ is also set. Route $r_i$ requires then a more constrained state for its command. However, the property of monotonicity can be easily checked through a static analysis. To do so, one can simply read sequentially the application data and check separately each condition. Furthermore, applying the notion of monotonicity to the set of itineraries instead of routes can also be done. □

**Proposition 2.** *Considering only pairs of routes is sufficient to verify the safety of an interlocking based on the application data format described previously provided that they are monotonic.*

*Proof.* We have to prove that all the requirements can be verified by using at most two routes. An issue can occur if the first route is not properly set, such a case only requires routes taken separately and is then trivially proved, or if the command or activation of another route interacts with components already locked for the first route. We need to prove that considering two routes is sufficient to detect all of these issues. Let us consider $C$, the set of all the components, either physical or logical, of the station and $C_i \subseteq C$, the set of components used or locked by Route $r_i$. Let us take two arbitrary routes, $r_1$ and $r_2$. There are two possible situations:

- $C_1 \cap C_2 = \emptyset$: the two routes have no component in common and are then completely disjoint. No issue can happen between them.
- $C_1 \cap C_2 \neq \emptyset$: the routes have at least a component in common. If the interlocking allows both routes to be set at the same time, an issue can happen.

Any issue can be represented as an intersection between such sets. An intersection is formed by at least two routes. Two routes are then sufficient to detect any safety issue provided that commanding a third route will not relax $C_1$ or $C_2$ by releasing some components thereafter. According to Proposition 1, the application data must be monotonic to avoid that. In this case, testing only all the pairs of routes is thus sufficient to cover all the conflictual scenarios. □

This kind of assumption is also considered in [15] where the verification is limited to two trains. Algorithm 1 presents how we performed the verification by considering all the pairs of routes. The `command` and `activate` instructions (lines 5 and 7) correspond to the requests defined in the application data, like Listings 1.1 and 1.4. The bidirectional locking request (Listing 1.3) is also done through `command` instruction. They return `True` if the request is fulfilled and `False` otherwise. Furthermore, if they are accepted, all the attached actions modifying the station state are executed. `move` instruction (lines 20 and 23) moves a train to the next track segment as defined by the points state. If a point is misplaced,

the train will either derail or pursue its movements until it leaves the station.

First, each pair of routes are considered (lines 1-2). The goal is to move a train $t_1$ from the origin of a route to its destination (lines 10-28) and for each position of $t_1$, we will try to command and to activate another route (lines 12 and 17). We also try to command $r_2$ directly after that $r_1$ has been commanded (line 6). Such a case can happen in real situations. If $r_2$ is successfully commanded and activated (line 18), we move a train $t_2$ until it reaches the destination of the route (lines 19-22). When a particular position of $t_1$ has been tested, $t_1$ goes to its next position (line 23) and the interlocking will try to release all the locked components (lines 27-28). Releasing conditions are described in the application data such as in Listing 1.5. Through the iterations on the positions of $t_1$, we memorize the fact that the other route, $r_2$, has been commanded or activated (lines 12 and 17). Indeed, because of the succession of release actions, the command and the activation can occur at different moments during the route life cycle. When a pair of routes has been entirely tested, the station is reinitialised (line 29) in order to have an empty station before testing the next pair. It is done through **reinitialise** instruction which releases all the locked components and removes all the trains of the station.

*Detection of issues* Requirement (1) is tested after each movement of $t_2$ by testing that its position can never be the same as $t_1$ (lines 21-22). Requirement (2) is tested each time $r_2$ has been commanded. If the current position of $t_1$ is a point, the point cannot move after the command of $r_2$ (lines 14-16). It is done by comparing its state with its previous one through the operator `previous`. Requirements (3) and (4) are tested on lines 24 and 25. If $r_1$ cannot be activated (lines 8-9), we consider that we have a fluidity issue because no other route is already activated (not presented as a requirement).

*Time complexity* Each pair of routes must be tested, as well as all the possible configurations of positions between two trains. We have thereby the theoretical bound $\mathcal{O}(r^2t^2)$ with $r$ the number of routes and $t$ the number of track segments. The verification of Braine l'Alleud Station took **148 seconds** on a MacBook Pro 2.6 GHz Intel Core i5 processor and with a RAM of 16 Go 1600 MHz DDR3 using a a 64-Bit HotSpot(TM) JVM 1.8 on Yosemite 10.10.5.

## 4 Experiments

Several kinds of errors have been introduced in the application data in order to test the adequacy of our algorithm and all of them have been successfully detected in Braine l'Alleud:

- Incorrect or missing conditions on a route command (Listing 1.1).
- Conditions missing for releasing a component (Listing 1.5).
- Route activation not consistent with the related route command or condition verifying the vacancy of a track segment is missing (Listing 1.4).

**Algorithm 1:** No conflictual pair of routes

```
1  for r1 ∈ ROUTES do
2      for r2 ∈ ROUTES such that r2 ≠ r1 do
3          place a train t1 at r1.origin
4          place a train t2 at r2.origin
5          r1.isCommanded ← command r1
6          r2.isCommanded ← command r2
7          r1.isActivated ← activate r1
8          if not r1.isActivated then
9              return False
10         while t1.position ≠ r1.destination do
11             if not r2.isCommanded then
12                 r2.isCommanded ← command r2
13             if r2.isCommanded and not r2.isActivated then
14                 for p ∈ POINTS such that t1.position = p do
15                     if p.state ≠ previous(p.state) then
16                         return False
17                 r2.isActivated ← activate r2
18             if r2.isCommanded and r2.isActivated then
19                 while t2.position ≠ r2.destination do
20                     move t2
21                     if t1.position = t2.position then
22                         return False
23             move t1
24             if t1.position ∉ TRACK_SEGMENTS then
25                 return False
26             remove t2 from S
27             for c ∈ COMPONENTS such that c.isLocked do
28                 release c
29         reinitialise S
30 return True
```

- Bidirectional locking not properly locked (Listings 1.3 and 1.4).

In order to analyse the scalability of our algorithm, we perform three experimentations. Firstly, we compare the execution time required to verify different numbers of routes in the station. A complete verification requires to consider all the possible routes. Indeed, limiting the number of routes only produces a partial verification. Secondly, in addition to Braine l'Alleud (17 tracks segments and 32 routes) we test our algorithm on a smaller instance, Nameche (13 tracks segments and 14 routes), and a larger one, a subpart of Courtrai (19 track segments and 70 routes). Finally, we compare our method with the approach of Busard et

al. [13] that have performed a model checking verification of Nameche. Figure 2 recaps the execution time of the different experimentations. Let us notice that the $y$-scale is logarithmic. As we can see, our algorithm runs faster ($\approx 4$ orders of magnitude for 14 routes) than the model checking approach, even for larger instances and more routes. Furthermore, the algorithm scales well for larger instances: a verification of all the routes is performed in less than 3 minutes for Braine l'Alleud and in less than 16 minutes for Courtrai. The experimentations have been performed on the same computer as in the previous section.

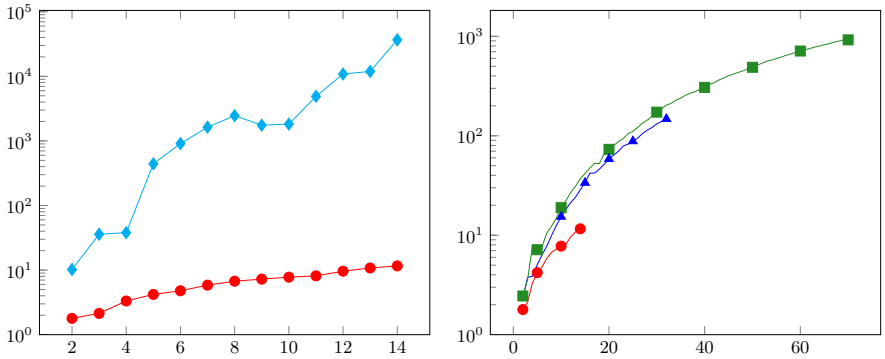


**Fig. 2.** Execution time (in seconds) in function of the number of routes in Nameche (●), Braine l'Alleud (■) and Courtrai (▲) by using our algorithm and the model checking approach of Busard et al. [13] for Nameche (⧫).

## 5 Conclusion

Much research has been carried out in order to automatically verify the correctness of an interlocking system. Up to now, most of it tackles the problem with a model checking approach or, more recently, using a discrete event simulation. Both of them have some limitations. On the one hand, model checking suffers from the state space explosion problem, and on the other hand, simulation does not provide sufficient guarantees that the system is correct. In this paper, we proposed another approach. The idea was to use our knowledge of the system not only to model it, but also to design the verification algorithm. Concretely, we implemented a dedicated polynomial algorithm that can verify the safety of a medium size station in less than three minutes and that can scale on larger stations provided that an assumption of monotonicity hold. We also shown their validity by introducing several errors in the application that were successfully detected. The method proposed in this paper only deals with the verification of safety. Availability properties, stating that the trains will always progress in the station, are not considered. However, whereas Standard EN50128 [3] strongly recommends the use of exhaustive methods for the verification of safety, the verification of availability can be based on non exhaustive methods as statistical model checking [16]. Both methods are complementary and a full verification

of an interlocking can then be based on a hybrid approach using the dedicated algorithm for the safety and statistical model checking for the availability.

## References


1. Cribbens, A.: Solid-state interlocking (ssi): an integrated electronic signalling system for mainline railways. In: IEE Proceedings B (Electric Power Applications). Volume 134., IET (1987) 148–158
2. Theeg, G., Anders, E., Vlasenko, S.: Railway Signalling & Interlocking: International Compendium. Eurailpress (2009)
3. CENELEC, E.: 50128. Railway applications-Communication, Signaling and Processing Systems-Software for Railway Control and Protection Systems (2011)
4. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal modeling and verification of interlocking systems featuring sequential release. In: Formal Techniques for Safety-Critical Systems. Springer (2014) 223–238
5. Winter, K.: Model checking railway interlocking systems. In: Australian Computer Science Communications. Volume 24. (2002) 303–310
6. Eisner, C.: Using symbolic model checking to verify the railway stations of hoorn-kersenboogerd and heerhugowaard. In: Correct Hardware Design and Verification Methods. Springer (1999) 99–109
7. Huber, M., King, S.: Towards an integrated model checker for railway signalling data. In: FME 2002: Formal MethodsGetting IT Right. Springer (2002) 204–223
8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. Springer (2012)
9. Winter, K., Johnston, W., Robinson, P., Strooper, P., Van Den Berg, L.: Tool support for checking railway interlocking designs. In: Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55, Australian Computer Society, Inc. (2006) 101–107
10. Winter, K.: Optimising ordering strategies for symbolic model checking of railway interlockings. In: Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. Springer (2012) 246–260
11. Cappart, Q., Limbrée, C., Schaus, P., Legay, A.: Verification by discrete simulation of interlocking systems. In: 29th Annual European Simulation and Modelling Conference 2015, ESM 2015. (2015) 402–409
12. Limbree, C., Cappart, Q., Pecheur, C., Tonetta, S.: Verification of interlocking systems using statistical model checking. arXiv preprint arXiv:1605.06245 (2016)
13. Busard, S., Cappart, Q., Limbrée, C., Pecheur, C., Schaus, P.: Verification of railway interlocking systems. In: Proceedings 4th International Workshop on Engineering Safety and Security Systems, ESSS. (2015) 19–31
14. Anunchai, S.: Verification of railway interlocking tables using coloured pertri nets. In: Proceedings of the 10th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. (2009)
15. Moller, F., Nguyen, H., Roggenbach, M., Schneider, S., Treharne, H.: Defining and Model Checking Abstractions of Complex Railway Models Using CSP||B. In: Hardware and Software: Verification and Testing. Volume 7857 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 193–208
16. Cappart, Q., , Limbrée, C., Schaus, P., Quilbeuf, J., Traonouez, L.M., Legay, A.: Verification of interlocking systems using statistical model checking. arXiv preprint arXiv:1605.02529 (2016)