Verification of railway interlocking systems

Quentin Cappart

Thesis confirmation step

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)
Louvain School of Engineering (EPL)
Université catholique de Louvain (UCL)
Louvain-la-Neuve
Belgium

Examining board

Prof. Pierre **Schaus**, *Supervisor*	UCL/ICTM, Belgium
Prof. Yves **Deville**	UCL/ICTM, Belgium
Prof. Charles **Pecheur**	UCL/ICTM, Belgium
Prof. Axel **Legay**	INRIA, France

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 because of the state space explosion problem.

In this thesis, we investigate new verification methods aiming to deal with this problem. More concretely, we introduce methods such as random simulation, statistical model checking and dedicated algorithm that were until now never applied for verifying interlocking systems. The relevance and performance of these methods are also analysed through different realistic stations of the Belgian railway network.

Acknowledgements

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

Foreword

Railway operators are faced with competition from road, air and maritime transport. They need to improve in terms of the globalisation of traffic and the interoperability between operations and infrastructure which are more complex than in other modes. To deal with this issue, Walloon Region of Belgium initiated in April 2014 INOGRAMS project. The main goal of this project is to maintain the competiveness of Wallonia's railway industry in the face of other transportation means. Given the large scope of this project, it is divided into seven work package.

I do my thesis in the context of this project. More specifically, I work on the first work package. Its goal is to propose innovative solutions for easing the future development of new interlocking systems. As the development of such systems must follow the highest safety requirements, the same rules must apply for our solutions. For this project, I collaborate with several companies and universities such as Alstom, Cetic, UMons and UNamur. Together we develop a tool which can be used to automatically generate a new kind of application data and verify their correctness. By correct we mean that it will never cause any safety of availability issue. My main contribution is related to the verification part. I investigate how the verification can be performed through different methods.

Contents

Chapter 1

Introduction

1.1 Research goals

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 are computerised systems composed of a generic software taking as input data, called application data, describing the actions that the interlocking must take for each situation that can occur in a particular station [1]. 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 [2]. Although the generic software is developed in accordance with these requirements, the reliability of an interlocking is also dependent 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. Beyond the safety, an interlocking must also ensure that no train will be stopped too long in the station in order to maintain the availability of the station. It is why availability, or fluidity, properties must also be considered. Most of the time, the verification of the application data, as well as its validation, is performed through testing on a physic simulator that reproduces the environment of the interlocking. This process is thus costly and error prone. Moreover manual testing does not cover all the scenarios that could possibly end-up in a unsafe situation.

To overcome this lack, research has been carried out in order to

improve the verification of the application data correctness. Most of it is based on model checking [3]. 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 issue are formalised. Finally, the model checker verifies that no reachable state of the model violates the 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.

Verification of railway interlocking system is a critical concern. Moreover, without optimisation or specific improvements, model checking cannot be used for verifying large stations. The goal of my research is to improve this verification. For that, I investigate several methods in order design a new verification process which can be used for verifying stations of any size and which can be fully automated.

1.2 Overview of the contributions

This section presents all the contributions that have been developed through this thesis. Their complete description will follow later in this document. The contributions can be classified into two different groups: contributions related to data analysis and contributions related to verification.

Contributions related to data analysis and preprocessing Different information such as the interlocking behaviour or the infrastructure are required to perform the verification. However, these information come from different data sources and have a format not directly exploitable. It is why one of the first tasks was to extract and analyse these data and to convert it into an adapted format. This group includes all the contributions related to the utility tools that we developed to perform

such tasks.

The first contribution is a tool designed to automatically parse application data expressed on Solid State Interlocking (SSI) format [4]. The output obtained can be used for several purposes and different kinds of modelling. For instance, it has been used in this thesis for designing a simulator of the interlocking behaviour. Furthermore, it has also been used in other works by Busard et al. [5] and Limbree et al. [6] in order to build a model which can be verified using different model checking methods.

Following the same idea, the second contribution is a parser tool extracting the topology of a station from a data source based on railML [7]. The simulator, as well as [5, 6] is also built using this translator.

Contributions related to verification This group includes the new verification methods that we introduced during this thesis. Our global contribution is a framework and a methodology allowing an user to automatically verify the correctness of an interlocking from its application data and the topology of its station. Several contributions related to different parts of the framework have been realised:

- A model instancing the behaviour of the application data and the topology of a particular station. The model presented here is designed in order to allow its simulation by a discrete event simulator and its verification using Bounded Linear Temporal Logic (BLTL).
- The formalisation of the safety properties defined by Busard et al. [5] in BLTL. Such a logic is used in order to have the possibility to determine when the simulator must stop its processing.
- The introduction of an availability property that an interlocking must face in order to ensure that no train would be stuck in a station. Such a property has also been formalised in BLTL.
- A Discrete Event Simulation (DES) engine which can be run on the top of the previous model. This tool contains several advanced features such as the possibility to stop a simulation at any state and

to replay it later. A command line interface as well as a graphical user interface have also been developed in order to facilitate its utilisation and its visualisation.

- The utilisation of Statistical Model Checking (SMC) methods such ad Monte Carlo estimation, Chernoff's bound and importance splitting algorithm, in order to verify the model running with the simulator engine.
- A polynomial dedicated algorithm verifying that an interlocking will never cause derailments or collisions provided that an assumption of monotonicity hold. It can also verify that each train will reach the correct destination.
- An executable tool instantiating the model, its simulation, the requirements and the different verification methods.

1.3 Publications

The work and methods presented on this thesis have already been presented in several publications:

- A first version of the model and its verification with a discrete event simulation approach have been published and presented at 29th European Simulation and Modelling Conference in October 2015 [8]. This paper describes how an interlocking can be verified using a random discrete event simulation and what is the interest of this approach compared to classical model checking approaches.
- ⟦UPDATE IF ACCEPTED⟧ An extended version of the model and its verification by statistical model checking have been submitted at *Formal aspect of computing* Journal. This paper deals with the drawbacks of [8] and improves the verification thanks to statistical model checking methods.
- ⟦UPDATE IF ACCEPTED⟧ Another extension of the model has been performed. Concretely, we added the bidirectional locking mechanism and the differentiation between a route command and a route activation. Furthermore, a polynomial dedicated algorithm verifying that the interlocking will never cause any safety issue

provided that some assumptions hold have been introduced. This work has been submitted at 35th International Conference on Computer Safety, Reliability and Security in September 2016.

1.4 Outline

With the exception of the introduction, the content of this thesis is divided into five other chapters. They are organised as follows:

- Chapter 2 gives the background related to the railway interlocking domain on which this thesis is based. The structure of an interlocking, its behaviour, its input data and how signalling engineers can use it are described inside. Three case studies exploited in this work are also introduced.
- Chapter 3 presents how we modelled an interlocking system. The architecture of the model, its performances, its limitations and the hypothesis done are detailed.
- Chapter 4 deals with the verification of an interlocking system. First, the intrinsic difficulties of an automatic verification are stated. The state of the art methods dealing these issues, as well as their limitations, are then described. After that, new verification methods not yet used for interlocking are introduced and analysed. Finally, a toolbox taking over the verification methods developed through this thesis is presented.
- Chapter 5 presents the software implemented during this thesis. This software encompasses the different methods, principles and algorithms presented in this document.

Finally, the last chapter summarises the work done through this thesis and sketches possibilities of future work.

Chapter 2
Interlocking principles

2.1 Context

A railway interlocking is an arrangement of systems that prevents conflicting train movements in a station. It is more specially a signalling subsystem that controls physical components of a station before allowing a train through the station. To do so, the interlocking collect information about the occupation of the track layout and about the movable elements such as the points. It then evaluates this information and can permit or refuse train movements by setting signals on a proceed or a stop state.

Basically, an interlocking has three main functions which can be split into three levels [1]:

- **Operational level**: it includes the interface between the human signaller performing the request and the machine.
- **Interlocking level**: it includes the functions required to decide if the request performed by the signaller can be accepted or not, and to do the proper actions consequently.
- **Element control level**: it mainly includes the functions required to transmit information between the components.

Over the years, the progression of interlocking technology has not stopped to grow up. Existing technologies can be categorized into four groups:

- **Human interlocking**: all the functions (performing the request, checking its satisfiability and moving components) are performed by a human. It is the oldest mechanism for an interlocking.

- **Mechanical interlocking**: the signaller controls the interlocking through mechanical levers which are connected with each other. Information transmission to the physical components is done by wires.
- **Electric interlocking**: the signaller controls the interlocking through electric buttons. The physical components are controlled electrically through a relay technology.
- **Computer based interlocking**: it is the technology that is currently mostly used in the world. It is also called electronic interlocking. All the functions are performed electronically through a computer hardware and software.

Table 2.1 recaps the four forms of interlocking technologies and how the three functions are performed.

	operational	interlocking	element control
human	-	human	human
mechanical	mechanical levers	lever frame	wires
electric	electric buttons	relays	electric
electronic	monitor/keyboard	hard/software	electronic

Table 2.1. Interlocking technologies and technical application of the functions in the three levels [1].

This thesis deals with computer based interlockings. Such 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, implying that no accident will occur, is the most important aspect to consider when designing an interlocking. For this reason, European Railway Agency has edited strict safety norms in an effort to harmonize the signalling principles and rules at the European level [2, 9]. Although the generic software is developed in accordance with these requirements, the safety of the train traffic relies also on the correctness of the application data.

Currently, the application data are prepared manually and are thus subject to human errors. For example, some prerequisites to the clearance

of the home signal of a route can be missing. This kind of error can easily be discovered by a code review or a static analysis [10]. However, errors caused by concurrent actions, like route commands, are much harder to spot. In this case, the combination of possible concurrent actions explodes quickly and testing manually all the combinations is impracticable. As testing all the possible scenarios is impossible, the manual validation of the application data relies on a relaxed verification process performed on three steps:

1. **Functional tests**: they ensure that the system responds properly to the commands issued by the controller. Those tests are performed by the expert who wrote the application data.
2. **Safety tests**: they check that each command is tested and that all the conditions that are supposed to impact the command are tested in all their possible values. Those tests are prepared and carried out by an independent tester.
3. **Reviewing**: the application data are finally reviewed by the engineer in charge of the project.

During this process, all the anomalies are traced in a bug management tool and must be fixed before the interlocking is commissioned. This 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.

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 [4] 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. When a train is entering in a station, a signalman assigns a route to the train by performing a route request. The interlocking will process the request, decide if it can be accepted and perform then some actions in order to set it.

The rest of this section states the components used by a route based interlocking, presents how they are formalised in SSI, explains their behaviour and illustrates it on a case study.

2.2 Components of an interlocking

Let us first present our case study: Braine l'Alleud Station. A representation of its topology with its related components is shown in Figure 2.1.

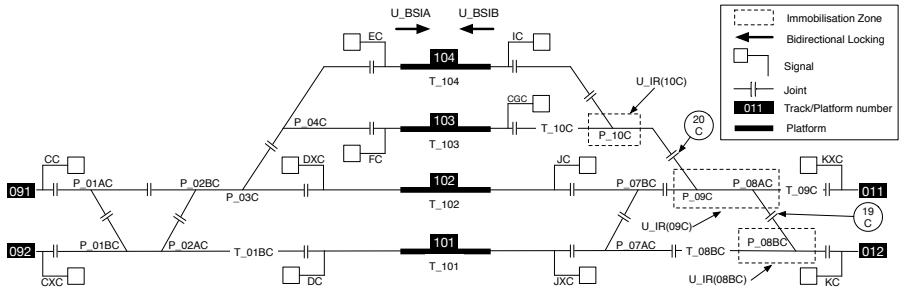


Figure 2.1. Layout of Braine l'Alleud Station.

This figure recaps the component types that are used in our model. On the one hand, there are the **physical components** of the track layout:

- The **tracks** (e.g Track `101`) are the railway structure where trains can move. A track can be a **platform** if the trains can stop on it in order to pick up passengers.
- The **track segments** (e.g `T_01BC`) are the portions of tracks where the trains 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). They are also called railway switches.
- The **signals** (e.g. `S_CXC` or simply `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, 12 points and 12 signals. The physical components are controlled and monitored by a single 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 on a proceed state if this action will not cause any safety issue.

On the other hand, there are the **logical components**. As previously said, a **route** is the path that a train is supposed to follow inside a station. It is generally named according to its origin and its destination place. Signals are often used as references 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 the 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 the 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.
- The **immobilisation zones** are the variables materialising the immobilisation of a set of points. When they are locked, their attached points cannot be moved. For instance, Immobilisation zone `U_IR(09C)` prevents Points `P_08AC` and P_09C to be commanded.
- 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 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.

2.3 Solid State Interlocking

Solid State Interlocking (SSI) refers together to the first computer based interlocking systems, and to the software used to develop these systems. Its development started in the 1970s by British Railways in collaboration with GEC (now Alstom) and Westinghouse. It quickly becomes one of the most widely used interlocking systems, especially in Western Europe and countries of the British Commonwealth [1]. The software is structured into two parts:

- The **generic software** which serves as an interpreter for the application data and carries out system functions like communications. This part does not change from one station to another and can then be verified once and only once. The development and the validation of that generic software follow the highest safety rules applicable to the domain.
- The specific **application data** which express all interlocking functions. They are written in a data language designed to be used by signalling engineers. Unlike the generic software, the application data are specific to a station and must then be verified for each station considered.

The ability of the interlocking to avoid critical situations, like train collisions, relies on the safety level achieved by the combination of the generic software and of the application data. Application data describe then the behaviour of an interlocking system instantiated to a specific station. They are divided into several configuration files. Only some of them are considered in this work:

- All the variables and structures used in the application data are declared in several **.id** files. Table 2.2 presents these variables with their possible values.

Component	SSI variable	Possible values
Subroute	`U_`*id*	`f` (free) / `l` (locked)
Immobilisation zone	`U_IR(`*id*`)`	`f` (free) / `l` (locked)
Bidirectional locking	`U_BSIA(`*id*`)` `U_BSIB(`*id*`)`	`f` (free) / `l` (locked)
Route	`R_`*id*	`s` (set) / `xs` (unset)
Point	`P_`*id*	`n` (normal) / `r` (reverse) + `cf` / `c` / `cd`
Track segment	`T_`*id*	`c` (clear) / `o` (occupied)
Signal	`S_`*id*	`stop` / `proceed`

Table 2.2. Variables with their possible values expressed in SSI format.

Each component has a unique identifier *id* and can have two values except for the points which encompass two information: a position (normal or reverse) and a status. The status refers to the life cycle of the points: free to be commanded (`cf`), commanded (`c`), and finally commanded and directed (`cd`). Their state is then an aggregation of both information. For instance, `P_01BC cdn` means that Point `P_01BC` has been commanded and directed to its normal position. Concerning the bidirectional locking, two variables are used for this mechanism. Les us define these components and their values with an EBNF syntax [11]:

⟨*component*⟩ ::= Subroute | UIR | UBSIA | UBSIB | Route | Point | Track_segment | Signal

⟨*value*⟩ ::= f | l | s | xs | c | o | cfn | cfr | cn | cr | cdn | cdr | stop | proceed

- **PRR.dat**: it defines the conditions that must be satisfied in order to accept a route request. The actions that the interlocking must perform when the request is accepted are also defined inside.

The instructions and expressions described in this file follows this grammar:

⟨*PRR_file*⟩ ::= {⟨*route_request*⟩}

⟨*route_request*⟩ ::= **Q*(**Route**) if** {⟨*condition*⟩}
then {⟨*action*⟩} [⟨*UBSI_expression*⟩]

⟨*UBSI_expression*⟩ ::= **if** (UBSIA | UBSIB) f
then (UBSIA | UBSIB) l

⟨*condition*⟩ ::= ⟨*component*⟩ ⟨*value*⟩

⟨*action*⟩ ::= ⟨*component*⟩ ⟨*value*⟩

- **PFM.dat**: it defines the conditions that must be ensured before moving a point.

⟨*PFM_file*⟩ ::= {⟨*point_request*⟩}

⟨*point_request*⟩ ::= ***Point**N* ⟨*condition*⟩
***Point**R* ⟨*condition*⟩

- **FOP.dat**: it defines the necessary conditions for releasing logical components after they have been used.

⟨*FOP_file*⟩ ::= {⟨*subroute_release*⟩ | ⟨*UIR_release*⟩ |
⟨*UBSIA_release*⟩ | ⟨*UBSIB_release*⟩ }

⟨*subroute_release*⟩ ::= Subroute f **if** ⟨*condition*⟩

⟨*UIR_release*⟩ ::= **if** UIR l **then if** ⟨*condition*⟩
then ⟨*action*⟩

⟨*UBSIA_release*⟩ ::= UBSIA f **if** ⟨*condition*⟩

⟨*UBSIB_release*⟩ ::= UBSIB f **if** ⟨*condition*⟩

- **OPT.dat**: generally speaking, it describes the life cycle of the routes from their command to their releasing. We use it to know when the start signal of a route can be set at a proceed state.

⟨*OPT_file*⟩ ::= {⟨*route_activation*⟩}

⟨*route_activation*⟩ ::= **if** ⟨*condition*⟩ **then** ⟨*action*⟩
else ⟨*action*⟩

Let us mention that the grammars presented here only cover a subset of the application data. Indeed, for the sake of simplicity, we refined it in order to formalise only the elements used in our analysis. Detailed explanations about the components and the expressions are provided in the next sections. The other configuration files serve different purposes such as the communication between several interlockings and are not related to safety or availability.

2.4 Interlocking behaviour in SSI

With both the physical and logical components, a route based interlocking controls the train traffic by monitoring the station, commanding routes, activating them, locking components and releasing them. In this section, we explain how a SSI interlocking manages these actions. All the possible actions and their underlying conditions are described in the application data. To illustrate the behaviour, let us consider the scenario where a train is coming from Track `012` and has to go to Platform `103` in Braine l'Alleud:

- Firstly, when the train is waiting at Signal `KC`, the interlocking will verify whether the request for Route `R_KC_103` can be granted. Listing 2.1 presents the request according to the application data of Braine l'Alleud.

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

Listing 2.1. 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 (lines 3-4) and if some immobilisation zones are not locked (line 5). If all the conditions are satisfied, `R_KC_103` is set (line 6), the points are controlled to the requested position (lines 7-8) and some components like the immobilisation zones (line 9) or subroutes (line 10) are locked. At this step, Route `R_KC_103` is set, or commanded, but not yet activated. Indeed, its start Signal `KC` 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 2.2 illustrates such conditions for Point `P_08AC`.

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

Listing 2.2. Conditions allowing Point P_08AC to move.

- Directly after the acceptance of the request described in Listing 2.1, 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 2.3.

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

Listing 2.3. Bidirectional locking request for 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,
3            P_10C cdn,
4            U_IR(08BC) l, U_IR(09C) l, U_IR(10C) l,
5            T_08BC c, T_09C c, T_10C c, T_103 c,
6            U_BSIA(103) f
7     then   U_BSIB(103) l,
8            S_KC proceed
```

Listing 2.4. Request for activating Route R_KC_103.

Listing 2.4 states that R_KC_103 can be activated only if the points are commanded and detected in the requested position (lines 2-3), if the immobilisation zones are thoroughly locked (line 4), if there is no train on some track segments (line 5) and if the bidirectional locking for trains coming from right to Platform 103 is free (line 6). The route activation results in locking the paired bidirectional locking (line 7) and in setting Signal S_KC on a proceed state. 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 2.5 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 2.5. Conditions for releasing Subroute U_20C_CGC.

This process describes the life cycle of a route and how it is managed by the interlocking.

2.5 Station topology in railML

The application data describe the behaviour of an interlocking applied to a specific station but contain no information about the track layout. However, the correctness of an interlocking is also related to its consistency with the track layout. It is why a reliable data source describing the track layout, or the **topology**, is necessary.

The topology can be encoded in different ways. A common approach is to represent it using a computer-aided design such as AutoCad [12]. Different information such as the track lengths or the geographic position of components can be described inside. It is the method currently used in Belgium. However, a graphical representation has some shortcomings. Given that the information is presented graphically, the processing of the schema and its integration into a verification model require manual works and can then hardly be automated.

Another way to represent the topology is to use a structured language. With this kind of representation, information can be automatically processed and used for different purposes. Based on a Extensible Markup Language (XML) structure, **railML** [7] is a markup language specialised for the railway domain. It was conceived to give a universal support for information which can be used for any applications related to the railway field. It is used by several companies such as Alstom, Siemens, Bombardier, Thales or Toshiba. As any markup language, railML is structured with schemas. There are gathered into four main schemas:

- **Infrastructure**: it describes the railway network topology and contains information about the physical components.
- **Timetable and Rostering**: it contains all the information about the timetables and the schedules.
- **Rolling stock**: it takes over all the information about the vehicles used (length, height, weight, etc.)
- **Common**: it encompasses the information not included in the other schemas.

Only the infrastructure schema is related to the topology. However, railML does not provide yet all the information required to represent the whole topology. For instance, information such as the positions of joints are missing. To overcome this lack, we enriched railML with new elements. We call this extension **railML+**. Figure 2.2 summarizes the elements that we use from the infrastructure schema as well as their hierarchy. Explanation of the different schemas are provided thereafter.

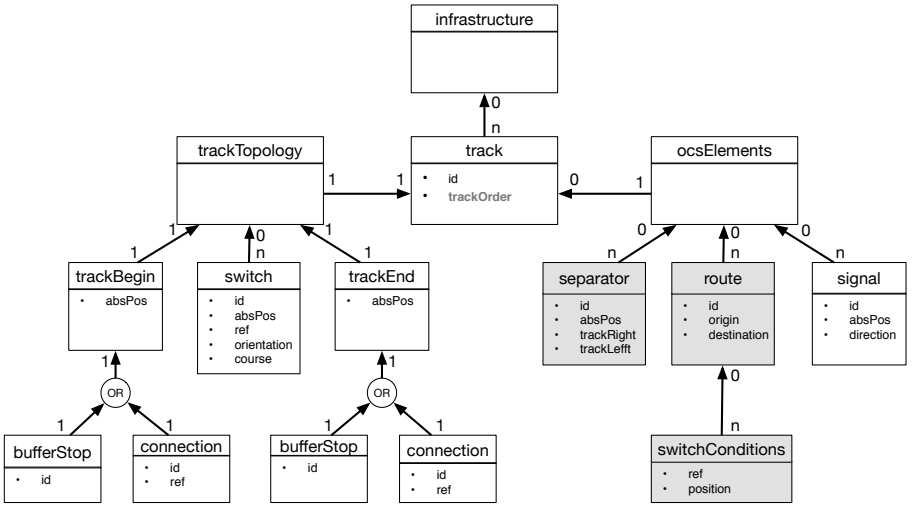


Figure 2.2. railML infrastructure schema enriched with new elements (in grey).

track An infrastructure has a set of tracks. Each of them is characterised by a unique identifier (`id`) and a relative position between the other tracks (`trackOrder`). In Braine l'Alleud, the track order varies from 0, for Track `101`, to 3 for Track `104`.

trackTopology Each track has one and only one topology (1 to 1 relation). This subschema encompasses different physical components of the tracks.

trackBegin It represents the beginning of a track. It is characterised by an absolute position (`absPos`).

trackEnd It represents the end of a track. It is also characterised by an absolute position (`absPos`).

switch It represents the definition of a point. A `trackTopology` schema can have several points on it (0 to n relation). Each of them is characterised by a unique identifier, an absolute position, the identifier of its linked component (`ref`), and finally its physical disposition (`orientation` and `course`). Figure 2.3 presents the four possible dispositions of a point.

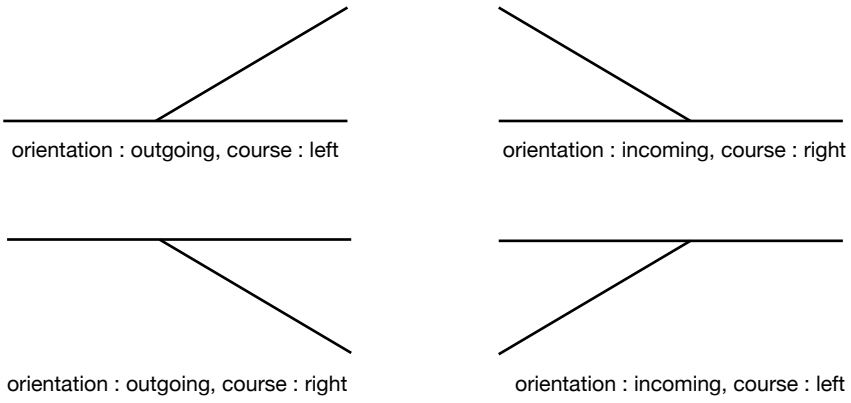


Figure 2.3. The fourth possible physical dispositions of a point.

For instance, Point `P_01BC` as an `incoming left` disposition while Point `P_02AC` has an `outgoing left` disposition.

bufferStop It is one of the two possible extreme elements (`trackBegin` and `trackEnd`) for a track. It represents the border with another infrastructure. For instance, Braine l'Alleud has four `bufferStop`: two on the left of Signals `CC` and `CXC` and two on the right of `KC` and `KXC`. It is characterised by a unique identifier.

connection It is the second possibility for a track extreme element. It models its connection with another component, typically a point. For instance, Track `103` is connected on the left with Point `P_03C` and on the right with Point `P_09C`.

ocsElements This subschema encompasses several physical and logical components.

signal It corresponds to the signals. They are characterised by a unique

identifier, an absolute position and a direction (`up` or `down`) defining the direction in which the signal must apply. In Figure 2.1, Signal `CC` is oriented upside whereas `KXC` is oriented downside.

separator It corresponds to the joints. Each of them has a unique identifier, an absolute position and also the name of the track segment located at the left (`trackLeft`) of the joint and at its right (`trackRight`). This subschema, as well as the following, is not included in the basic version of railML but only in the extension railML+.

route It correspond to a route as defined earlier. Each of them has an identifier, an origin (`origin`), and a destination (`destination`).

switchConditions A route is also defined by the position of all the points crossed by the route in order to reach its destination. This subschema encompasses this information. It contains for each point, a reference (`ref`) to the points as defined in the `switch` schema, and their required position (`position`) in order to form the proper path from the origin of the route to its destination. For instance, Route `R_CXC_102` requires Points `P_01BC`, `P_02AC` and `P_02BC` to be set at the normal position and Point `P_03C` to be set at the reverse position.

With these schemas, railML+ can thereby be used in order to specify the track layout of a station with more flexibility than the graphical approaches. It is why we propose here to use railML+ as a reference for the topology.

2.6 Case studies

Three cases studies are analysed in this thesis: Braine l'Alleud Station, Namêche Station, and LK7 area of Courtrai Station. The choice of these case studies is not fortuitous. We chose stations of different size in order to have a realistic and broad overview of the different stations in Belgium. If the proposed methods can be applied on the three cases studies, they can theoretically also be applied for each Belgian station.

2.6.1 Station of Namêche

Namêche is a Belgian city located near Namur in Wallonia. It has a small sized station composed of 4 tracks, 13 track segments, 7 points, 7 signals, 14 routes, 7 immobilisation zones and 26 subroutes. The bidirectional locking mechanism is not used in this station. The entire station is controlled by a single interlocking. A representation of its track layout is shown in Figure 2.4.

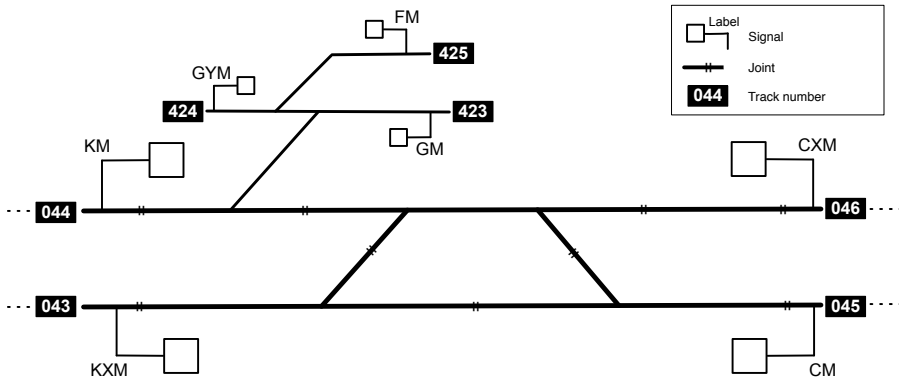


Figure 2.4. Layout of Namêche Station.

2.6.2 Station of Braine l'Alleud

Already introduced in this document, Braine l'Alleud is a Belgian city located in the center of the country in Walloon Brabant Province. Its station is on the direct line (Line 124) between Charleroi and Brussels. It is a medium sized station composed of 4 tracks, 17 track segments, 12 points, 12 signals, 32 routes, 10 immobilisation zones, 48 subroutes and 4 bidirectional locking mechanisms. The entire station is also controlled by a single interlocking. A representation of its track layout is shown in Figure 2.1.

2.6.3 Station of Courtrai

Courtrai, or Kortrijk, is a Belgian city located in the west of Belgium in Flanders. It is considered as a large station. The entire station is controlled by three interlockings that communicate together. Here, we are only interested in a subpart (LK7 Area) of Courtrai controlled by a single interlocking. A representation of the track layout of LK7 is shown in Figure 2.5.

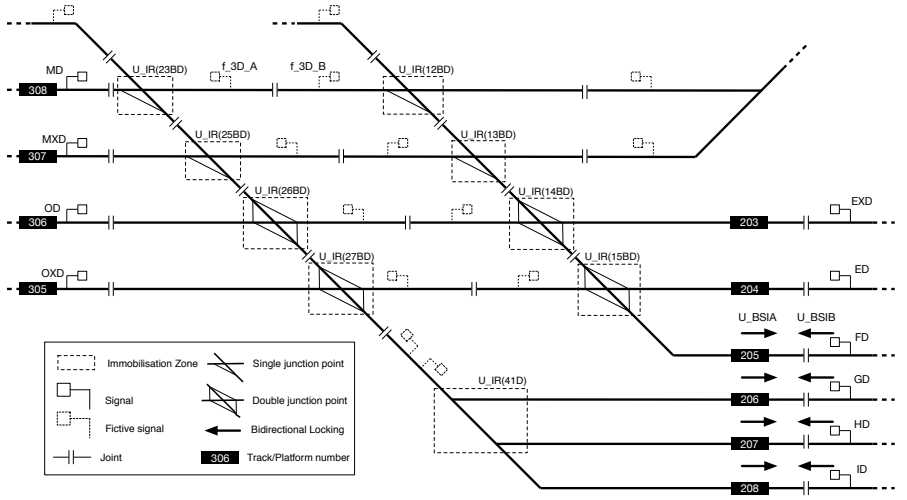


Figure 2.5. Layout of LK7 area of Courtrai Station. [[schema to be modified]]

It is composed of 6 tracks, 19 track segments, 26 points, 24 signals, 70 routes, 9 immobilisation zones, 72 subroutes and 4 bidirectional locking mechanisms. Concerning the signals, 14 of them are fictive. A fictive signal is a signal that is not materialised in the track layout but that has a representation in the application data. Unlike Namêche and Braine l'Alleud, a route in Courtrai does not correspond to a complete itinerary for a train. When a train is entering into the station and must go to a particular destination, it is not done with one route but with a set of routes that must be commanded sequentially. For instance, if a train must go from `S_ED` to `S_MD`, two routes must be used sequentially, one from `S_ED` to `S_f_3D_B` and a second one from `S_f_3D_A` to `S_MD`. In

Courtrai, an itinerary is then a sequence of routes.

Chapter 3

Model of an interlocking system

3.1 General approach

The previous chapter described the principles of an interlocking system and how it is used to regulate the train traffic. It also introduced the problematic of application data verification and the lacks of the current situation in Belgium. This chapter presents how an interlocking system can be modelled and how its behaviour can be reproduced on a computer in order to automatically verify the application data correctness through several verification approaches. The structure of this chapter follows a top-down hierarchy. First, a global overview of the entire verification process is presented. Each step composing the approach is then separately detailed.

Our verification process is divided into several steps:

1. Generating a model of an interlocking by combining its application data and the track layout of its station. This is performed by two translators that parse and aggregate both data sources into a single model.
2. Creating a model representing a real train traffic in order to analyse how the interlocking model react to it.
3. Creating a model mimicking the actions that a human signalman can perform when trains arrive in the station.
4. Stating and formalising all the requirements that an interlocking system must satisfy in order to ensure a safe and fluid train traffic.

5. Reproducing the interlocking behaviour under a realistic train traffic. It is done through a simulator. A set of traces summarizing the different actions that occurred during the simulation as well as the states reached are then obtained.
6. Analysing the traces obtained and verifying that none of them contains states violating the requirements.

This entire process is gathered into a single framework implemented in Scala [13]. This language is multi-paradigm and supports the object oriented [14] and functional paradigm [15]. It is also compatible with Java language and its libraries [16]. The data flow diagram presented in Figure 3.1 resumes the main steps of the approach.

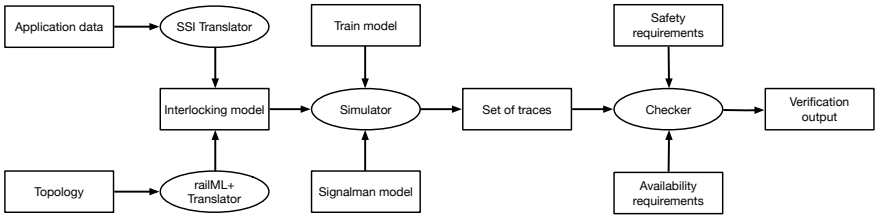


Figure 3.1. Step of the verification approach.

More details on the first steps are provided in the next sections. Steps related to verification are detailed in the next chapter.

3.2 SSI translator

The first step is to translate the input data into an exploitable format. This section is dedicated to the translation of SSI application data presented on Sections 2.3 and 2.4. As illustrated on Figure 3.2, the translation is done in two steps: the elaboration of grammars and the parsing.

Grammar elaboration Before translating the application data, grammars defining the format of the application must be specified. Such grammars are used by the parsing tool in order to capture the structure

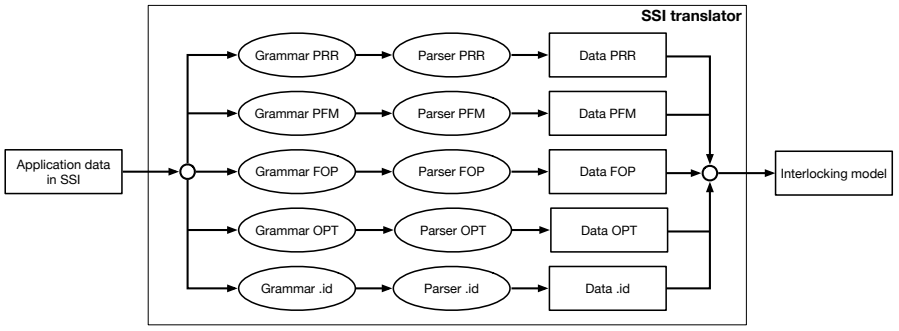


Figure 3.2. SSI translator architecture.

of SSI language and then to know how it can be translated. As SSI data are described into several configuration files having each a particular, several grammars are required. The grammars used are presented in Section 2.3. According to Chomsky's hierarchy [17], such grammars are context-free [18].

Definition 3.1 (Context free grammar)**.** *A context free grammar is a grammar where every production rule follows the pattern* $X \rightarrow \alpha$ *where,* X *is a non terminal symbol and* α *a sequence of terminal and/or non terminal symbols.*

Context free grammars are often used for specifying programming languages because there exist efficient algorithms such as Early [19] or CYK [20] for their parsing. Furthermore, more efficient algorithms (Packrat [21], recursive descent [22], etc.) can be used provided that the considered grammar satisfies some other properties.

Parsing Once the grammars are specified, the next step is to use them for parsing the application data. Because several grammars are used, we designed a particular parser per grammar. However, some structures, as the the specification of a variable, have the same syntax in every file and are then common for every grammar. We use the inheritance mechanism [23] to take advantage of it. Figure 3.3 presents the inheritance diagram of our parser. Each parser is implemented as a class and inherits from `Parser SSI` which encompasses the common expressions of each file.

Furthermore, the parsers are designed to skip all the data non used in our model.

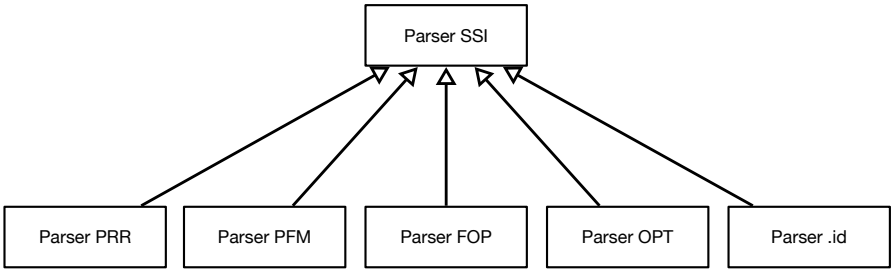


Figure 3.3. Inheritance diagram of the parser.

The parser uses a combinatory parsing technique [24]. It has been implemented using the combinator parser (`util.parsing.combinator`) package of Scala [25].

3.3 railML+ translator

As for the application data, the data related to the topology must also be translated in order to build the interlocking model. Its translation, illustrated in Figure 3.4, follows the same process.

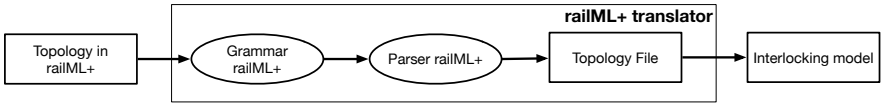


Figure 3.4. railML translator architecture.

Grammar elaboration As presented in Section 2.5, the topology is expressed in railML+. Its grammar follows then an XML pattern. All the considered data are detailed in Figure 2.2.

Parsing Tthere exist diverse algorithms for parsing XML data [26]. Our implementation is based on XML scala library (`scala-xml`) which

follows the standard SAX parser from Java library [27]. As output, the parser create a topology file which encompasses three information about the station:

- Characteristics about the physical components.
- How the physical components are connected together.
- All the possible routes in the station with their path determined by the position of the points.

3.4 Interlocking model

This section presents the structure of the interlocking model. Figure 3.5 gives an overview of it. Three components are involved: the configuration of the interlocking, the topology of the station, and the interlocking system itself. All of them are implemented as an object.

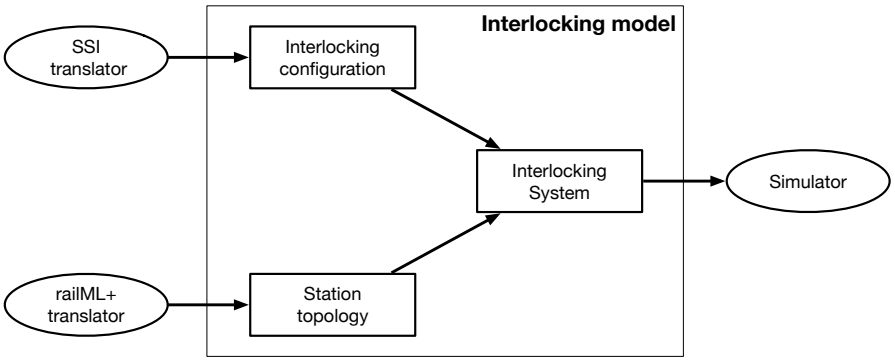


Figure 3.5. Architecture of an interlocking model.

Interlocking configuration This object is built from the parsing of the application data. It includes several attributes:

- A list for each kind of interlocking component (immobilisation zone, bidirectional locking, track segment and subroute). Each component is represented as an object with two attributes: a

unique identifier and its state, as presented in Table 2.2. The `track segment` objects include also a set of references of trains that occupy the segment.

- A syntax tree for each configuration file parsed. Such trees include all the information parsed that are required for the interlocking model.

Station topology This object is built from the parsing of the topology file. We model the station using graph theory [28]. Graph theory has already been deeply studied for modelling complex networks [29]. There are numerous examples of concrete applications as Internet [30], social networks [31] and transportation networks [32]. It has also been applied for railway signalling [33]. We model a station as an undirected graph. Points, signals, buffers stops and joints are represented by nodes and their connections by edges. Furthermore, each edge belongs to a particular track segment. The implementation is done using Jung library [34].

Interlocking system The interlocking system is only an aggregation of the two previous structures. It is used to reproduce the interlocking behaviour described in Section 2.4 and to apply it on the considered station. It is given as input for the simulator thereafter.

Assumptions done Furthermore the interlocking model is based on several assumptions:

- Signals can only have two states: stop and proceed.
- The start signals of routes move to the proceed state immediately after that the route has been activated.
- There is no distinction between a normal route and a shunting route. A shunting route is a route that the train can follow only with a low speed in order to join it with another train.
- All the physical components are perfect and there is no failure of them.
- When a train moves through a joint, the modification of track segment is automatically recorded.

- Communication between several interlockings is not considered.
- The level crossing control and its interaction with the routes is not modelled. A level crossing is an intersection between a railway line and a road.

These assumptions are mainly done in order to keep the model as simple as possible and then limit the state space size. Such assumptions are also proposed in other works [5, 35, 36].

3.5 Train model

The previous section introduced how we modelled an interlocking system. Let us remember that the goal of an interlocking system is to ensure a safe and fluid train traffic. However, until now the train traffic is not yet considered. This section describes how we model trains. As proposed in different works [5, 35, 36], our modelling is based on several assumptions:

- Trains follows properly the signalling principles. For instance, they do not overrun signals on a stop state.
- Speed and length of trains are abstracted.
- Trains just occupy one and only one track segment at a time.
- Trains can stop instantly.
- Trains have a perfect behaviour. They have no faulty component and have no train failure.
- Trains only enter in the station from buffer stops.

Furthermore, the model contains three attributes:

- An **identifier**: each train has an unique identifier.
- A **position**: it defines the track segment where the train is located.
- A **direction**: it defines the direction of the train. It has two possible values, `up` or `down`.

Furthermore, trains only move from track segment to track segment until they reach the end of the station. As we will see in the next section,

this model is used in order to build a mutable entity which can be simulated. The interlocking model is then tested with a traffic composed of trains modelled in this way.

3.6 Signalman model

The next step is to model the signalman. Generally speaking, the signalman is the human who sets the signals and the points in order to control the train traffic. For modern route based interlockings, the signalman has the responsibility for performing route request when trains enter into the station.

In our case, we model the signalman as an object without attributes but which performs route requests for trains waiting at a start signal of a route.

3.7 Simulator

Simulation is a science used everyday and applied to a large number of fields such as weather forecasting [37], transportation [38], logistic [39] or healthcare facilities [40, 41]. It has also been applied to the railway field. For instance, Sogin et al. [42] analyse through simulation the effects of higher speed passenger trains in freight networks. Furthermore, the company OpenTrack provides a railway simulation tool [43] to verify the capacity of a railway network, the feasibility of the schedules, collect statistics about running times, etc. Generally speaking, a simulation can be defined as *an imitation of a system* [44]. The main force of simulation is that it allows the study of various systems without building the system, thus saving precious time, cost and effort. Real life includes myriads of systems. Checkland [45] identified four classes:

- **Natural systems**: they are the systems created from the origins of the universe (weather, movements of planets, etc.).
- **Designed physical systems**: they are the physical systems created from the result of a human design (production facility, transportation network, etc.).

- **Designed abstract systems**: they are the abstract systems created from the result of a human design (mathematics, literature, etc.).
- **Human activity systems**: they are the systems modelling human activities with their interactions (political system, communication, etc.).

All of these systems can be modelled and then be simulated. In our case, we are interested by *interlockings*, a specific designed physical system. The previous section described how this system can be modelled. This section presents how its simulation can be performed. Firstly, generalities about simulation are presented. The principles of our simulation is then described. Finally, the integration of this simulation and its application for our railway model is detailed.

3.7.1 Simulation taxonomy

A simulation can be performed in a plenty of ways. According to the considered system, some kinds of simulation can be more adapted than others. Sulistio et al. [46] identified several inherent characteristics of simulation:

Presence of randomness A simulation can be either **deterministic** or **stochastic** according to the presence of randomness or not. In deterministic simulations, there is no randomness. It means that for a given input, the output will always be the same. In stochastic simulations, the output can be different for a same input.

Presence of time Time can be considered in the simulation or not. A **static simulation** imitates a system at a particular point in time while a **dynamic simulation** imitates the system with its progression in time.

Time slicing When time is considered, it can be represented in two ways. Firstly, there are the **continuous simulations** which consider an infinity of values through a bounded time interval. Secondly, they are the **discrete simulations** which discretise the time into instants. A time interval T from 0 to 4 seconds can for instance

be discretised into 0, 1, 2, 3, 4 seconds and then produce 5 values. For the same interval, a continuous simulation would produce an infinity of values.

Time interval For discrete simulations, the management of time intervals can be done in different ways:

- **Time driven simulation**: the simulation progresses by fixed time increments. Figure 3.6 illustrates this process.

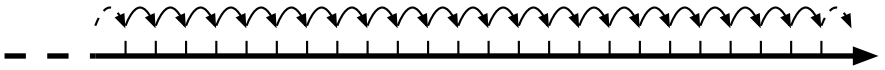


Figure 3.6. Behaviour of a time driven simulation.

- **Event driven simulation**: the simulation progresses by irregular time increments according to the execution instant of the events. An event can be defined as a modification on the system state. If there is no event planned for a particular instant, it will then not be considered. Figure 3.7 illustrates this process.

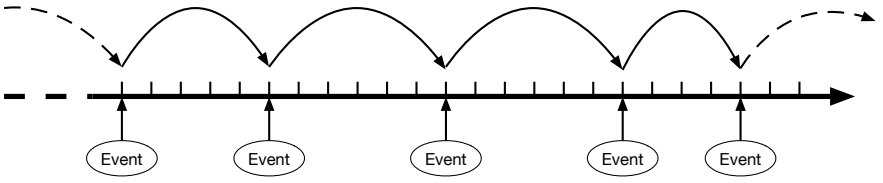


Figure 3.7. Behaviour of an event driven simulation.

- **Trace driven simulation**: the simulation progresses by reading a set of events collected from another environment and that has been previously executed.

Figure 3.8 illustrates such taxonomy. The grey boxes represent the choices made for our system. As we can see, our simulation follows a probabilistic dynamic discrete event pattern. However, the *dynamic*

and *probabilistic* attributes are often implicit in computer simulation. It is why this kind of simulation is often shorted as a **discrete event simulation** (DES) [47].

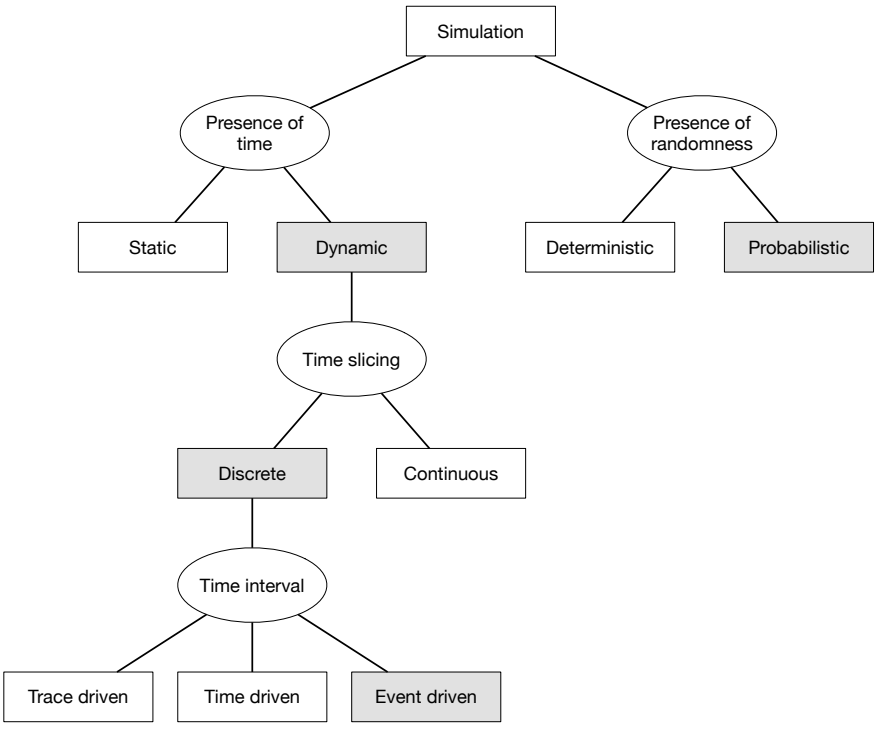


Figure 3.8. Simulation taxonomy with the choices made for our system (in grey).

3.7.2 Principles of discrete event simulation

This section describes the principles of a discrete event simulation and presents how it can be implemented. A discrete event simulation involves five kinds of components:

Entities They are the active objects on which the simulation is applied. Each entity is characterised by a list of attributes with a specific value. This list is called the **entity state**. The term *active*

means that the value of attributes can change through time. For instance, a train can be an entity characterised by a direction and a position. Figure 3.9 illustrates how we represent Train `IC732` entity.

Using a programming language supporting the object oriented paradigm such as Java or Scala, an entity schema can be implemented as a *class* and an entity as an *object*.

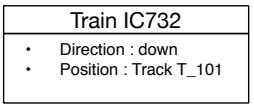


Figure 3.9. `Train IC732` entity.

Events They define the actions that can change the entity state and which can generate other events.

Definition 3.2 (Event)**.** *In a simulation, an event is an action changing the system state. It is characterised by two attributes:*

- *The action.*
- *The time at which the action must be executed.*

For an event `e`, `e.process` is the action and `e.time` is its execution time. For instance, we can consider an event which applies on a `train` entity and which moves the train to its next position. Figure 3.10 illustrates this behaviour. A guard can also be present for events. It defines conditions that must be satisfied for activating the event. For instance, let us consider a situation where a train is waiting in front of a signal. The event can be triggered only if the signal is on its proceed state.

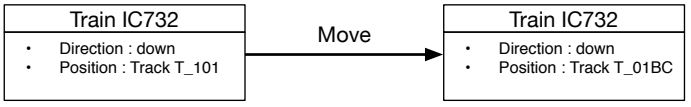


Figure 3.10. `move` event applied to Train `IC732`.

Clock It states when the events must be executed. Unlike a continuous simulation where events can occur during a time period, the discrete simulation requires each event to occur at a particular instant. Furthermore, it is used to model the progression of the simulation through time. For an event based simulation, this mechanism is often organized as a priority queue [48], sorted by event time. Jones [49] proposes several implementations for it. With such a data structure, the next event to be executed is the event with the closest event time from the actual time of the simulation.

Pseudo random number generator As previously said, this simulation is probabilistic. It means that two simulations from identical inputs can have a different progression. Such a mechanism requires to generate randomness. However, generating pure random variables with a computer is not feasible [50]. For this reason, the use of a Pseudo Random Number Generator (PRNG) [51] is required.

Definition 3.3 (Pseudo Random Number Generator)**.** *A PRNG is an algorithm used for generating from an initial value called the seed, a sequence of numbers which looks like a sequence of numbers randomly picked up.*

For a given seed, a PRNG always produces the same sequence. The choice is the seed is then crucial. It is often determined by unpredictable parameters having a high level of entropy such as the exact timing of keystrokes and the movements of the computer mouse [52]. There exists a large variety of PRNG in the literature [53, 54, 55, 56], each having their own specificities.

Ending condition It states when the simulation must end. Typically, we end the simulation after a given number of iterations or after that a particular state has been reached. For instance, we can stop the simulation after that an accident has occurred.

Each of these components must be implemented in order to have a discrete event simulation engine.

3.7.3 Simulator architecture

The design of our simulator is then based on the discrete event paradigm. This section presents the architecture of the simulator and describes its different components. Firstly, Figure 3.11 gives an overview of this architecture. A description of each component is provided thereafter.

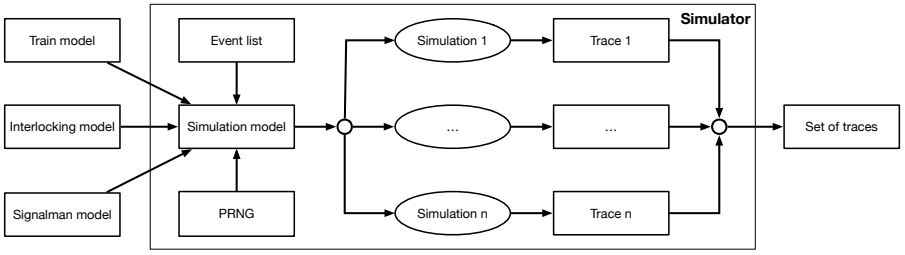


Figure 3.11. Simulator architecture.

Interlocking model The interlocking model is the model built from the aggregation of the application data and the station topology as presented in Section 3.4. It is used to infer a bunch of entities. Concretely each interlocking component is represented as an entity.

Definition 3.4 (Interlocking entity)**.** *An interlocking entity is an entity obtained from the interlocking active components.*

The exhaustive list of interlocking entities with their mutable attributes is as follows:

- The routes, on a `free`, `commanded` or `activated` state.
- The signals, on a `proceed` or `stop` state.
- The points, on a `normal` or `reverse` position.
- The subroutes, immobilisation zones and bidirectional locking, each of them on a `free` or `locked` state.
- The track segments, characterised by the number of trains on it.

Train model In the same way than the interlocking model, the train model is the model described in Section 3.5. The `train` entity if inferred from it.

Definition 3.5 (Train entity)**.** *A train entity is an entity obtained from the train model. It is characterised by two mutable attributes: a position (the current track segment occupied by the train) and a direction (`up` or `down`). Each train is identified by a unique identifier (`id`).*

Signalman model It is the model of the signalman performing the route requests.

PRNG Given that randomness is present in the simulation, a PRNG is thereby required. To do so, a linear congruential generator [57] with a 48-bit seed is used.

Definition 3.6 (Linear congruential generator)**.** *A linear congruential generator is an algorithm used in order to generate a sequence of pseudo random numbers. It is based on the following recurrence relation:*

$$X_{n+1} = (aX_n + b) \ mod \ m$$

where a, b and m are parameters. The next number in the sequence is then determined by its previous number.

An implementation is available in `java.util.Random` package for Java or Scala. This implementation can generate numbers with a uniform probability. Given that we use randomness only to generate different scenarios through simulations, we are not concerned by having a strict unpredictable sequence of number or a cryptographically secure generator [58]. The choice of the PRNG is then not a critical concern.

Event list The event list contains all the events that will be processed through the simulation. The simulation contains two kinds of events: events related to train movements and events related to interlocking actions.

Definition 3.7 (Train event)**.** *A train event is an event related to the actions that the trains perform from their arrival in the station to their departure.*

There are three train events:

- `addTrain(t,bf)`: this event adds a new train called t at the `bufferStop` called bf (Section 2.5). The station state is updated accordingly. Furthermore, it triggers two events: another `addTrain` event and a `moveTrain` event. The execution time of such events is determined by the PRNG.
- `moveTrain(t)`: this event moves Train t to its next position on the station and updates the station accordingly. Trains always move upside or downside according to their direction, from one track segment to the next one and follow the path defined by the points. If t is in front of a signal on a stop state, the train will not move. In any case, this event triggers one of the following event: another `moveTrain` or a `removeTrain` if the train has reached a `bufferStop`. The execution time is also determined by the PRNG.
- `removeTrain(t)`: this event is triggered by a `moveTrain` event only if Train t is in front of a `bufferStop`. Such an event removes t of the station and updates it accordingly.

With such events, each train has thereby its own queue of events going from `addTrain` to `removeTrain` with a sequence of `moveTrain`.

There are also the interlocking events.

Definition 3.8 (Interlocking event)**.** *An interlocking event is an event related to the actions that the interlocking can perform.*

An interlocking event can be one of the following types:

- `generateRoute(r)`: this event is periodically issued for the trains waiting at a start point of Route r. If the route command conditions are fulfilled, r is commanded and all the actions described in the request are executed as well as the related bidirectional locking request. The station state will then change. Otherwise, the request is discarded and no action is taken. Route command requests are described in the application data, as presented in Listings 2.1, 2.2 and 2.3.

- `activateRoute(r)`: this event is periodically issued when Route r is already commanded but not activated yet. If the conditions are fulfilled, r is activated (Listing 2.4). Otherwise, the request is discarded and no action is taken. Furthermore, after three unsuccessful activation requests for the same route, a `destroyRoute` event for r is triggered.
- `destroyRoute(r)`: this event performs a hard release of Route r and unlocks each component previously requested by the route.
- `releaseComponents`: this event tries to release every locked component on the station (Listing 2.5). If the conditions are not fulfilled, no action is taken. Such an event is periodically issued and before each train movement.

As for the train events, the execution time of all interlocking events is determined by the PRNG. Furthermore, their execution also triggers another event of the same type. Figure 3.12 illustrates a possible scenario.

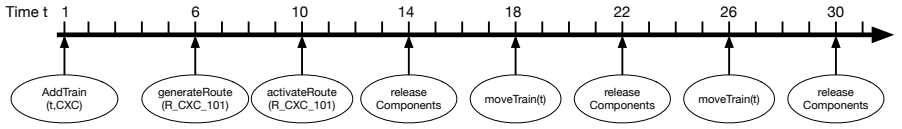


Figure 3.12. Possible scenario with the simulator.

The event list is ordered using a priority queue, sorted by event time. We use the default implementation proposed by the Scala API (`scala.collection.mutable`) which uses a heap data structure [59]. A priority queue contains two main instructions: `add(Event e)` which adds the event e inside the queue, and `pop()` which removes from the queue the event with the highest priority and returns it. Before the simulation, the event queue is fed with initial events.

Simulation model The simulation model is obtained from the aggregation of the interlocking model, the train model, the signalman model, the PRNG and the event list. It is the model which can be simulated. Concretely, it gathers the information of the inputs into a **simulation state**. The simulation state s_i is a description of the considered station at the i^{th} simulation step. It can change after each event which has

occurred during the simulation. For a simulation of n steps, a simulation state s_i with $i \in [1, n]$ is defined as

$$s_i : \left\langle nb, \sigma_p, \sigma_r, \sigma_s, \sigma_{uir}, \sigma_{ubsi}, \sigma_{track}, \sigma_{train} \right\rangle \tag{3.1}$$

where

- s_i is the state at the i^{th} step of the simulation.
- nb is the number of trains that have moved in the station so far. This variable is used to under approximate how many real days the simulation has covered. Indeed, by taking the extreme case of a busy station where there is an incoming train every minute all the day long, we can safely assume that the simulation has covered at least one real day when 1440 trains have moved through the station.
- σ_p : `point` $\rightarrow \{$`normal`, `reverse`, `default`$\}$ is a function defining the position of a point. The `default` state represents a point that is not positioned yet.
- σ_r : `route` $\rightarrow \{$`unset`, `commanded`, `activated`$\}$ is a function defining the state of a route.
- σ_s : `subroute` $\rightarrow \{$`free`, `locked`$\}$ is a function defining if a subroute is free or locked.
- σ_{uir} : `uir` $\rightarrow \{$`free`, `locked`$\}$ is a function defining if an immobilisation zone is free or locked.
- σ_{ubsi} : `ubsi` $\rightarrow \{$`free`, `locked`$\}$ is a function defining if a bidirectional locking is free or locked.
- σ_{track} : `track` $\rightarrow \mathbb{N}$ is a function defining the number of trains being on a track segment.
- σ_{train} : `train` $\rightarrow$ (`track`, $\{$`up`, `down`$\}$) is a function defining the current position of a train and its direction.
- `point`, `route`, `subroute`,`uir`, `ubsi`, `track` are the set of the interlocking components defined in the application data and `train` the set of trains in the station.

Furthermore, the simulation model also contains an event queue E and an ending condition *halt* expressed by means of variables of s. For instance, the ending condition can be $nb < 100$ if we want to stop the simulation after the departure of 100 trains from the station. More details about how we defined the ending condition is provided in the next chapter.

Simulation Once the simulation model is built, the next step is to simulate it. Algorithm 3.1 presents how the simulation is performed through a pseudo code. The algorithm takes as input a simulation model and returns the simulation trace of this model. While the ending condition is not satisfied (lines 4 to 8), an event is popped from the priority queue (line 5) and is then processed (line 7). The simulation state, as well as its time (line 6), is updated accordingly. Furthermore the new state is appended to the trace (line 8).

Algorithm 3.1: Discrete event simulation algorithm.

```
1 Input: a simulation model sm
2 Output: the trace t (represented as a list of states) of the simulation of sm
3 t ← sm.s_0
4 while sm.halt = False do
5 |   e ← sm.E.pop()
6 |   i ← e.time
7 |   e.process // update the simulation state
8 |   t ← t + sm.s_i
9 return t
```

The simulator has been implemented using the discrete event simulation package of OscaR [60], a Scala toolkit for solving operations research problems. This toolkit has similar functionalities as SimPy [61]. Furthermore, advanced features have been added to the simulator engine:

- The possibility to save a simulation state and use it as an initial state for new simulations.
- the visualisation of the simulation through a Graphical User Interface (GUI).
- The possibility for the user to interact with the simulation on the fly. Concretely, the user can trigger by himself the transitions, save

a state, load it, stop the simulation, and restart it.

- The possibility for the user to parametrise the simulation before its execution (PRNG seed, activation of the GUI, changing the ending conditions).

More details about these features are provided in Chapter 5 which presents the software implemented.

Resulting traces The trace is the output of the simulation. It recaps the simulation state for each step. For instance, a simulation of the scenario presented in Figure 3.12 produces the simplified trace described on Table 3.1. The checkmark ✓means that the event associated to the state transition has been accepted while ✗ means that it has been refused or that no action is taken. For instance, `releaseComponents` of s_{14} released no component. Each simulation provide a similar trace which encompasses all the simulation states as described in Formula (3.1).

Several simulations can be performed. In this case, each of them provides a different trace. The traces can be gathered into a set of traces for posteriori analysis.

	R_CXC_101	P_01BC	P_02AC	UIR_(01BC)	IC 442
s_0	unset	default	default	free	-
s_1 ✓	unset	default	default	free	(up, T_092)
s_6 ✓	commanded	normal	reverse	locked	(up, T_092)
s_{10} ✓	activated	normal	reverse	locked	(up, T_092)
s_{14} ✗	activated	normal	reverse	locked	(up, T_092)
s_{18} ✓	activated	normal	reverse	locked	(up, T_01BC)
s_{22} ✗	activated	normal	reverse	locked	(up, T_01BC)
s_{26} ✓	activated	normal	reverse	locked	(up, T_101)
s_{30} ✓	activated	normal	reverse	free	(up, T_101)

Table 3.1. Simplified trace of the scenario presented in Figure 3.12. ✓means that the event associated to the transition of state has been accepted, ✗ otherwise.

Chapter 4
Automatic verification

4.1 Motivation

As previously said, verification of railway interlocking system is a critical concern. Even is the interlocking generic software is developed in accordance with the highest safety requirements, it is not the case of the application data. However, the reliability of an interlocking is also dependant of the correctness of its application data.

Preparation of application data is still nowadays done by tools that do not guarantee the required level of safety. Most of the time, the verification of the application data, as well as its validation, is performed through testing on a physic simulator that reproduces the environment of the interlocking. This process is thus costly and error prone. Moreover manual testing does not cover all the scenarios that could possibly end-up in a unsafe situation.

Improving the verification process of the application data is an active field of research. There exist many methods an many ways of modelling. However, the state of the art methods have some shortcomings:

- Some of them are not fully automated. A human interaction is required at some steps of the process, often at the model elaboration which remains specific for each station.
- The verification does not scale for large stations and is then limited to small or medium sized stations. Indeed, the execution time or the memory consumption quickly become too important as the size of the station.

Our motivation is to design innovative methods dealing with both

issues. The previous chapter introduced how a model can be automatically inferred from an aggregation of two data sources. This chapter focuses on how a scalable verification can be performed. It is structured as follows:

- Firstly, the requirements that an interlocking must satisfy are formalised.
- Secondly, different common approaches, as model checking, are described.
- The new approaches that we designed are then described and studied through several experimentations.
- Finally, related works about interlocking verification are sketched.

Concerning the experimentations, all of them have been realised on a MacBook Pro 2.6 GHz Intel Core i5 processor and with a RAM of 16 Go 1600 MHz DDR3 using a 64-Bit HotSpot(TM) JVM 1.8 on Yosemite 10.10.5.

4.2 Definition of requirements

Before verifying that an interlocking is correct, we need to define what is exactly a *correct interlocking*. As previously stated, the goal of an interlocking is to ensure a safe and fluid train traffic in a station. It includes two notions:

- **Safety**: it ensures that the interlocking will cause no accident in the station in any situation.
- **Availability**: it ensures that every trains progress in the station and that the interlocking will not block them for too long.

As illustrated in Figure 3.1, the checker takes as input safety and availability requirements. However, they must first be expressed in a language supported by the checker. This section presents how such requirements can be formalised.

4.2.1 Safety properties

The general safety requirements for interlocking systems are described in detail by Tombs et al. [36]. They indicate that no **collision** and no **derailment** can occur in the station. Busard et al. [5] identified three safety properties related to these requirements:

1. A track cannot have two trains on it at the same time in order to avoid collisions (`no collision` property).
2. A point cannot move if there is a train on it. Otherwise it will derail (`no derailment1` property).
3. A point must always be set on a position allowing trains to continue their path. Otherwise, the trains will derail (`no derailment2` property).

Such properties can be expressed by means of Linear Temporal Logic (LTL) operators [62]. LTL is a logic used to encode formulae describing sequence of states. In addition to the logical operators, it includes temporal operators:

- **X**, for ne**X**t. $\mathbf{X}\phi$ means that Formula ϕ has to hold in the next state.
- **G**, for **G**lobally. $\mathbf{G}\phi$ means that ϕ has to hold in all the next states.
- **F**, for **F**inally. $\mathbf{F}\phi$ means that ϕ has to hold in at least in one of the next states.
- **U**, for **U**ntil. $\phi\mathbf{U}\psi$ means that ϕ has to hold until a state satisfying ψ has been reached.
- **R**, for **R**elease. $\phi\mathbf{R}\psi$ means that ϕ has to hold until and including a state satisfying ψ has been reached.

This logic is more expressive than invariants where the formulae must only be true in every state without considering sequences of states.

Furthermore, bounds can be introduced by enriching the LTL formulae with a Bounded Linear Temporal Logic (BLTL) formalisation [63]. BLTL is a logic typically used for defining the number of steps on which

a property must hold. In our case, BLTL is used for defining simulation bounds. Let us now formalise these properties.

`no collision` property Each track cannot have more than one train on it. It is formalised in Equation (4.1) for Track `T_01BC`. Each track has a similar condition. The variable n means that the property has to hold for n consecutive states.

$$\mathbf{G}_n\Big(\texttt{T\_01BC} \leq 1\Big) \tag{4.1}$$

`no derailment1` property If there is a train on a track segment having a point, the point cannot move. It is formalised in Equation (4.2) for Point `P_02AC`.

$$\mathbf{G}_n\Big(\texttt{T\_01BC} = 1 \implies \texttt{P\_02AC} = \mathbf{next}(\texttt{P\_02AC})\Big) \tag{4.2}$$

It can be translated as: *for n iterations, if there is a train on Track segment `T_01BC`, Point `P_02AC` must keep its position in the next state.*

`no derailment2` property If a train is coming from one of the two branches of a point, the point must be set in a position allowing the passage of the train. It is formalised in Equation (4.3) for Point `P_01BC`.

$$\begin{aligned}\mathbf{G}_n\Big((\texttt{T\_092} = 1 \wedge \mathbf{next}(\texttt{T\_092}) = 0 \wedge \mathbf{next}(\texttt{T\_01BC}) = 1)\\ \implies (\texttt{P\_01BC} = \texttt{normal} \wedge \mathbf{next}(\texttt{P\_01BC}) = \texttt{normal})\Big)\end{aligned} \tag{4.3}$$

It can be translated as: *for n iterations, if there is a train on Track segment `T_092` and if the train has left Track segment `T_092` and is now on Track segment `T_01BC`, then Point `P_01BC` must be set at its normal position. Each point and each branch has a similar condition.*

Together, these three properties constitute the safety requirements. They must always be satisfied in order to prevent the interlocking to cause accidents.

4.2.2 Availability properties

Beyond the safety, an interlocking must also ensure that no train will be stopped too long in the station in order to maintain its availability and the fluidity of the traffic. It is why availability requirements must also be considered. A train is blocked in the station if its assigned route is never activated. It directly yields to the following property:

1. A route could always be *finally* activated (`route availability` property).

Finally refers to the temporal operator: after any states of the system, there exists at least another state where the property is satisfied. It is formalised in Equation (4.4) for Route `R_CC_101`.

$$(\mathbf{GF})_n\Big(\texttt{R\_CC\_101} = \texttt{activated}\Big) \tag{4.4}$$

This property is sufficient to ensure the interlocking to respect the availability of the traffic.

4.3 Application data errors

This chapter deals with application data verification. It directly implies that such data can be erroneous. This section introduces several kinds of errors that can exist in the application data. A non exhaustive list is as follows:

(a) An incorrect or a missing condition for moving a point in a route command (Listing 2.1). It can lead the interlocking to command a route even in not safe situations.

(b) A point moved to a wrong position when commanding a route (Listing 2.1). It can cause a derailment in the station.

(c) A subroute, or another logical component (immobilisation zone or bidirectional locking), not properly locked when commanding a route (Listing 2.1). It can lead the interlocking to command other routes even if it is not safe.

(d) A missing condition for releasing a logical component (Listing 2.5). With such an error, the component could be released too early and lead to the command of unsafe routes.

(e) Irrelevant additional conditions for releasing a logical component (Listing 2.5). It can cause a perpetual locking of the component which will never be released. It will cause an availability issue.

(f) A non consistency between a route command and its activation (Listings 2.1 and 2.4). A route commanded risks then to never be activated.

(g) A missing condition verifying the vacancy of a track segment on a route activation (Listing 2.4). It can directly lead to a collision.

(h) A bidirectional locking not properly locked (Listings 2.3 and 2.4). It can lead to a head to head collision.

Such errors often lead to a safety or an availability issue. However, the application data are, by their design, robust. It means that an error inside them will not always lead to an issue because some internal controls are done. For instance, in Listing 2.1, the free condition for an immobilisation zone (`U_IR(08BC) f`) is checked twice: in this same listing, and also in Listing 2.2. Furthermore, the activation request for a commanded route also checks that the components have been thoroughly locked. It is why an error in the application data do not irremediably lead to a safety or an availability issue. The next sections present how the detection of issues can be performed. The reliability of the introduced methods is then tested with some experiments. For instance, some errors are introduced in the application data in order to see if they are thoroughly detected.

4.4 Model checking

A common approach for system verification is model checking [64]. The goal of a model checker is to verify if a system meets a set of properties by considering all the reachable states of the model representing the system. It is done in three steps. First, a model representing the system must be designed. Secondly, the requirements that the model must ensure are formalised. Finally, a model checker verifies that no reachable state of

the model violates the requirements. The main advantage of this method is exhaustiveness. It means that if a requirement is not satisfied, the model checker will always detect it.

Currently, model checking is one of the main approaches used for verifying interlocking systems. It has been deeply studied in the literature. For instance, Winter et al. proposes models, methods and improvements for the verification of control tables [65, 66, 35, 67]. The control tables provide the functional specifications for railway signalling interlockings and contain the key safety requirements. Mirabadi et al. [68] implemented a control table generator and a verifier of its correctness. Eisner uses symbolic model checking for verification of interlockings in the Netherlands. Symbolic model checking [69] consists in representing the states by sets and transition relations in order to explore more efficiently the state space.

Among these works, only few of them deal with the specific case of interlocking expressed in SSI format [5, 70]. However the time and the memory required to compute all the states of an interlocking in SSI are far more important than for boolean interlockings expressed with control tables.

Model checking suffers from the so called state space explosion problem [71]. 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. The verification is then restricted to small or medium sized stations [72]. Several techniques to limit this problem have been proposed. Winter et al. [35] suggest to keep the model as simple as possible by abstracting some parameters, such as the trains speed or length. Winter [67] also proposes several strategies to optimise the variable ordering. Several works advocate the utilisation of symbolic model checking [73, 74].

For the specific case of interlockings expressed in SSI, Busard et al. [5] proposed a NuSMV [75] model with customized model-checking algorithms based on operation on the Binary Decision Diagram (BDD) using PyNuSMV [76]. Huber et al. implemented a symbolic model checker

based on NuSMV with different optimisations like a dynamic variable re-ordering. However, even with such improvements, the verification still remains infeasible for large stations.

Figure 4.1 illustrates the state space and the model checking process. On this figure, the non conflictual states are represented by circles while the conflictual states are represented by stars. They are grey if the state has been visited through the search and red otherwise.

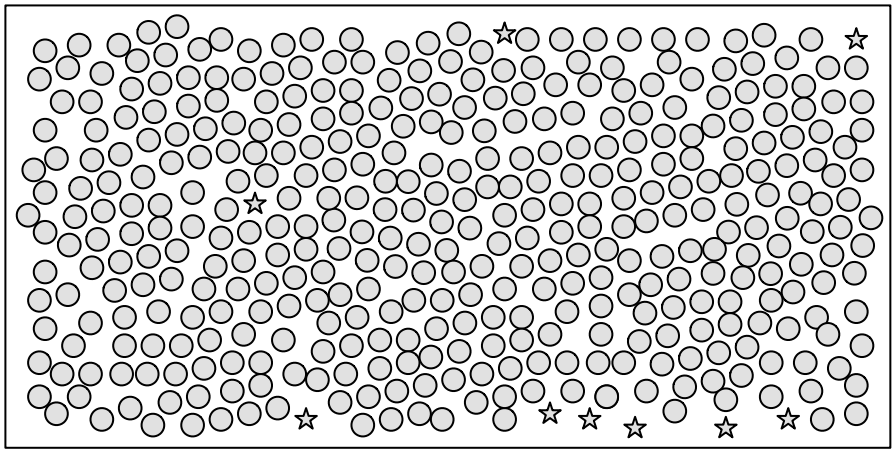


Figure 4.1. Verification by model checking. Circles represent the non conflictual states and stars the conflictual ones.

As we can see, all the states are explored and then all the possible conflictual states will then theoretically be detected. However, let us consider an interlocking managing 70 different routes as LK7 area of Courtrai (Figure 2.5). It yields at least 2^{70} different states. Currently, the state of the art model checking algorithms are not able to process such state space in a feasible execution time in practice.

Furthermore, even if safety requirements have been deeply considered, it is not the case of availability requirements which are expressed in LTL and then harden the verification.

4.5 Random simulation

In this section, we present a novel approach based on a random discrete event simulation which does not suffer from the drawbacks of model checking. Instead of considering the entire state space, the idea is to simulate the train movements and the behaviour of an interlocking as described in its application data and to observe if some issues have occurred. If no issue has occurred and provided that the simulation time was long enough, we can have a high expectation that the system is safe. Compared to model checking where all the states are considered even the ones that never occur in practice, the random simulation will only consider the cases which can potentially happen with a real interlocking.

Concretely, the main idea is to analyse the trace obtained from a simulation as explained in Section 3.7. Once the simulation is launched, we can observe the expected behaviour of the interlocking system as described by its application data and how it allows the trains to move through the station. The analysis of its behaviour is finally used to verify the correctness of the application data. The principle of this approach is illustrated in Figure 4.2. A simulation is initiated and is processed until a conflictual state has been reached. At this step, we know that the interlocking is incorrect.

However, this approach requires to define a **stop condition**. In other words, it means that we must define when the simulation must be stopped. If a conflictual state has been reached, we know that the interlocking is incorrect and we can then stop the simulation. However, if no conflictual state is reached, after how many steps can we stop the simulation ? For instance, if the simulation is stopped too early, we can have the situation of Figure 4.3 where the simulation has reached no conflictual state. In this situation, the interlocking is considered as correct even if it not the case.

Intuitively, if we extend the simulation time, we will have a higher expectation that the errors will be successfully discovered because more states are explored. The question is to determine after how much time the simulation can be stopped in order to have enough confidence that

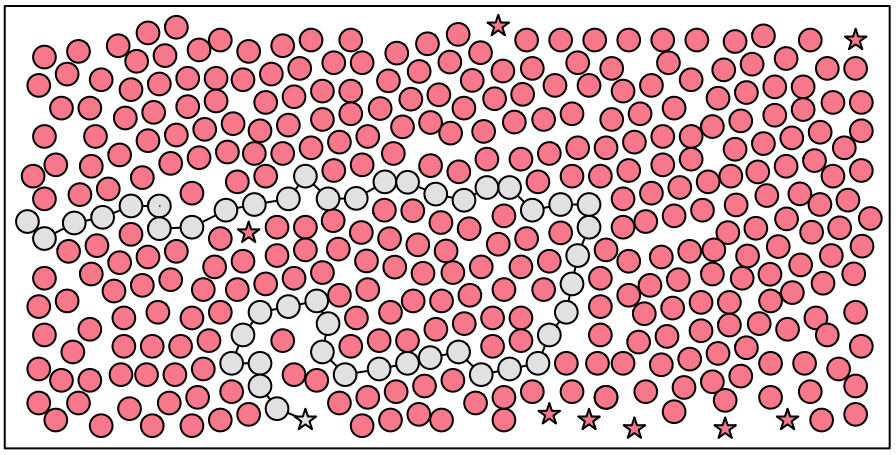


Figure 4.2. Successful verification by random simulation. States are grey if they have been visited through the search and red otherwise.

the system is correct. To do so, we proceed into several steps. Firstly, we need to define the correspondence between the simulation time and the real time. In other words, we must find how many real hours are covered by a simulation of a given execution time. Section 3.7 stated that a simulation has covered one real day when 1440 trains have moved through the station. With this information, the real time can be directly deduced. Table 4.1 shows the results for a simulation of 1 hour for Namêche, Braine l'Alleud and Courtrai.

Namêche		Braine l'Alleud		Courtrai	
# trains	real time	# trains	real time	# trains	real time
134291	93 days	16674	11 days	-	-

Table 4.1. Number of trains that have moved through Namêche, Braine l'Alleud and Courtrai after a simulation of 1 hour with its correspondence in real days.

Secondly, we can use the deduced real time to bound the simulation according to an arbitrary threshold. For instance, if no accident occurred after a simulation covering one thousand day, we can have a high expectation that the model is correct. According to the results of Table 4.1, a simulation of 11 hours is required in order to cover 1000 real days

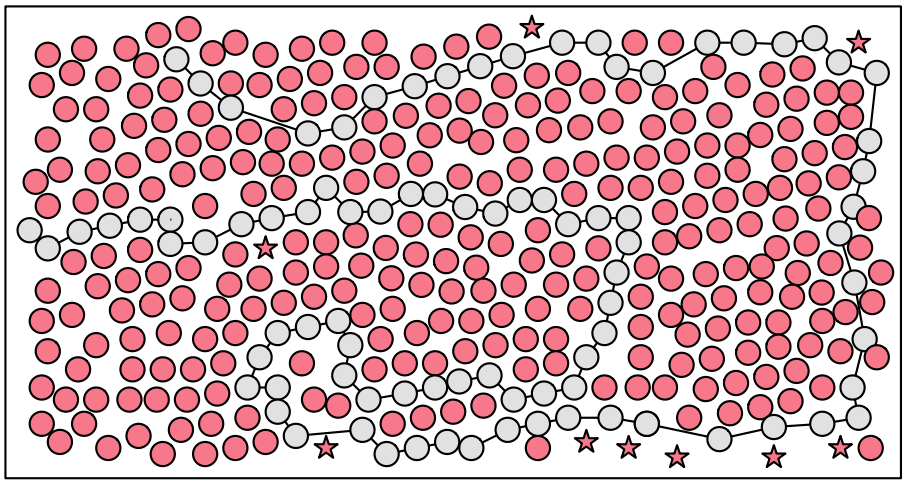


Figure 4.3. Unsuccessful Verification by random simulation. States are grey if they have been visited through the search and red otherwise.

in Namêche while 91 hours are required for Braine l'Alleud. Such an extrapolation can be done because the relation between the simulation time and the covered time increases proportionally. Concretely, doubling the simulation time will double the covered time.

The main drawback of this approach is that it does not provide enough guarantees that all the conflictual scenarios will be detected. A situation as the one presented in Figure 4.3 can then happen. Given the high safety level required for an interlocking, this method cannot be used without obtaining more confidence on the reliability of the simulation. For instance, the simulation time is currently chosen arbitrary. A better solution would be to determine the simulation time in order to obtain a sufficient safety level.

4.6 Statistical model checking

The previous sections introduced two approaches for system verification: model checking and random simulation. However, both approaches have drawbacks. Model checking suffers from the state space explosion prob-

lem while random simulation does not provide enough guarantees about the correctness of the system. This section proposes an intermediate approach, offering both the advantages of model checking and simulation.

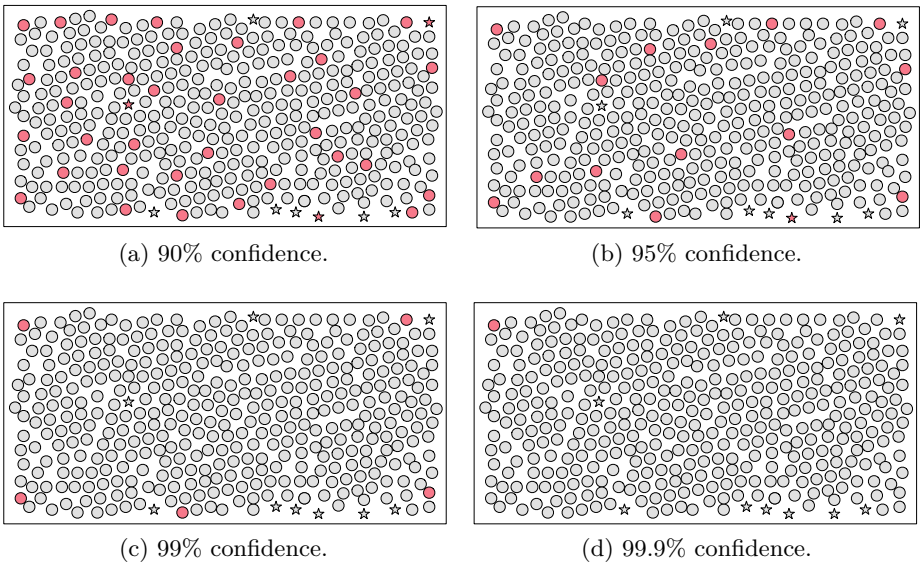
The limitation of random simulation is that only one simulation of an arbitrary period is performed. Our key idea is to perform instead several simulations and to observe through hypothesis testing and other statistical tests whether the results obtained provide a statistical evidence that the system is correct. Even if exhaustiveness is not obtained, we can still have a parametrizable confidence on the reliability and the availability of the entire system. A statistical model checker [77] can be used for that. The aim of **Statistical Model Checking** (SMC) is to approximate, in a controlled manner, the probability of satisfaction or violation of a property. Figure 4.4 illustrates the main idea of this concept. In this figure, different confidence thresholds can be obtained according to the reliability desired. Unlike classical model checking approaches where an exhaustive exploration of the state space is conducted, SMC only requires a sample of simulations.

SMC has already been used for verification of several applications, often stochastic [78], as biological [79], biochemical [80], electronic [81] or aircraft [82] systems. However, to the best of our knowledge, it has never been used yet for the verification of railway interlocking systems.

Improvements on the verification can be provided in many ways. This section describes how SMC can be used in order to gain more confidence on the verification performed by a random simulation. Several principles and algorithms are explained.

4.6.1 Monte Carlo estimation

Verification by random simulation is performed by a single simulation of an arbitrary period. However, this approach can be risky. Given its randomness, it is possible that the simulation becomes too specific and that it does not reflect the behaviour of the entire system with enough accuracy. A better idea would be to execute several simulations and to observe if the verdict is the same. With this improvement, if one or



(a) 90% confidence. (b) 95% confidence.

(c) 99% confidence. (d) 99.9% confidence.

Figure 4.4. Verification by statistical model checking for different confidence intervals. States are grey if they have been visited through the search and red otherwise.

several simulations do not provide the same result than others, it will be detected. For the verification, if only one simulation has detected an issue, we have a certitude that the system is incorrect.

This improvement can be provided by Monte Carlo method [83, 84]. The algorithm aims to estimate a probability γ of satisfying a property φ. The principle is to generate N random simulations $\rho_1, \ldots, \rho_N$ and to compute an estimation of γ. Equation (4.5) illustrates this estimation.

$$\tilde{\gamma} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\rho_i \models \varphi) \tag{4.5}$$

The term $\mathbf{1}$ is an indicator function that returns 1 if φ is satisfied in ρ and 0 otherwise. For instance, if one hundred simulations are performed and if no collision occurs, then the estimation $\tilde{\gamma}$ will be of 100%, which means that the `no collision` property is satisfied in the totally of the

simulations.

A confidence interval can also be obtained with Equation (4.6).

$$\left[\left(\tilde{\gamma} - z_\alpha \times \sqrt{\frac{\tilde{\gamma}(1-\tilde{\gamma})}{N}}\right), \left(\tilde{\gamma} + z_\alpha \times \sqrt{\frac{\tilde{\gamma}(1-\tilde{\gamma})}{N}}\right)\right] \tag{4.6}$$

The value z_α is the $(1 - \alpha/2)$ quantile of the standard normal distribution [85]. Intuitively, the more simulations are carried out, the stricter is the confidence interval.

4.6.2 Bound choice

A simulation requires to fix the number of iterations that must be executed. It is the simulation bound. This subsection analyses which values can be chosen for this bound. Intuitively, the bigger is the bound, the longer will be the simulation time, and the better will be the verification, but the execution time is also increased.

The bound can then theoretically be as high as possible. With Monte Carlo method, the choice of the bound can be turned in choosing the number of simulations. Indeed, under some assumptions, a simulation of m steps can be split into N simulations of $1/m$ steps. In other words, a simulation covering 30 days is identical to two simulations of 15 days and to three simulations of 10 days. We can then split a long simulation into several shortest simulations. However, this division can only be performed if at least one **complete scenario** occurs in each simulation.

Definition 4.1 (Complete scenario)**.** *A complete scenario is a scenario going from a train arrival to its departure in a station. The scenario is not complete if the simulation is stopped when the train is still waiting or moving into the station.*

The key idea through this assumption is to force the simulations to be able to generate any situation that can occur in the station. It can be done by fixing a lowest simulation bound. Indeed, if the bound is too low, the simulation time will be too short and some scenarios will not

be covered. It is why we require that the bound must be sufficient to determine a simulation time long enough to cover at least one complete scenario. For instance, a simulation covering one hour will not be sufficient because trains arriving in a station could still be in the station after one hour and causes an accident.

A lowest bound must then be fixed. In our case, we assigned a simulation time of one complete day. This value is chosen under the reasonable assumption that a train would not stay into the same station longer than one day. As explained in Section 3.7, the simulation has covered at least a complete day after the passage of 1440 trains.

4.6.3 Number of simulations

The next step is to determine how many simulations are necessary in order to have a strong enough confidence about the correctness of the system. Such information can be expressed in term of confidence intervals.

To do so, **Chernoff's bound** [86] is the method commonly used. From a confidence δ and a precision ϵ taken as parameters, this bound determines the required number of simulations (N) to perform in order to have a confidence interval on the estimation obtained by a Monte Carlo estimation. Chernoff's bound follows the relation described in Equation (4.7).

$$Pr(|\gamma - \tilde{\gamma}| < \epsilon) \geq 1 - \delta \quad \text{if} \quad N \geq \frac{ln(\frac{2}{\delta})}{2\epsilon^2} \tag{4.7}$$

Performing N simulations guarantees that the probability that a property is satisfied is included in the $(1-\delta)$-$[\tilde{\gamma} - \epsilon, \tilde{\gamma} + \epsilon]$ confidence interval. $(1-\delta)$ indicates the percentage of chance that the estimation obtained lies in the interval. In our case, we consider that the system is correct is it lies in a $(1-\delta)$-$[1-\epsilon, 1]$ interval.

Parameters δ and γ can then be chosen according to the reliability threshold desired. For instance, a 0.99-$[\tilde{\gamma} - 0.01, \tilde{\gamma} + 0.01]$ confidence interval on $\tilde{\gamma}$ with $\epsilon = 0.01$ and $\delta = 0.01$ can be obtained with 26 492

simulations of one day each. Furthermore, refining δ by a factor 10 will increase the number of required simulations by 11513 while refining ϵ by the same factor will multiply this number by 100. Knowing the execution time required to perform a 1-day simulation for each station from Table 4.1, we can deduce the expected execution time required to obtain such confidence intervals. A recap of the values obtained is presented on Table 4.2.

	Namêche	Braine l'Alleud	Courtrai
Number of simulations	26 492		
Real time covered	72 years		
1-day simulation exec. time	38.71 sec	327.27 sec	-
Total exec. time	12 days	100 days	-
Added exec. time for $\frac{\delta}{10}$	+5 days	+44 days	-
Added exec. time for $\frac{\epsilon}{10}$	$\times$100		

Table 4.2. Characteristics of the verification for a 0.99-$[0.99, 1]$ confidence interval using Chernoff's Bound.

The results of this table show that even poor confidence intervals (0.99-$[0.99, 1]$) for a small station require a consequent execution time.

4.6.4 Parallelisation

The theoretical results obtained from Chernoff's bound analysis show that considering a too strict confidence interval has the same drawbacks than model checking. Too many states must be explored and the execution time became quickly too important. However, a crucial advantage of simulation compared to model checking is that it can be easily parallelised without any overhead. Indeed, by Gustafson's law [87], the execution of N simulations on m processors will be m times faster than the execution of N simulations on a single processor. Even if model checking can also be parallelised [88, 89], it is a harder task and the parallelisation gain decreases with the number of processes.

Table 4.3 summarizes the number of processors required to perform in one hour the verification of our case studies in order to obtain a

(0.99-[0.99, 1]) confidence interval. These results show that it is possible to reach strong Chernoff's bounds in practice thanks to parallelisation. For comparison, the EC2 cloud computing services proposed by Amazon contains more than 30 000 cores [90].

	Namêche	Braine l'Alleud	Courtrai
1 hour	285	2408	-
12 hours	24	201	-
24 hours	12	101	-
10 days	2	10	-

Table 4.3. Number of processors required for performing the verification with a 0.99-[0.99, 1] confidence interval for different time periods.

4.6.5 Covering tests

⟦Results on this section must be updated for the new model⟧

A next drawback of simulation is that we have no guarantee that a particular scenario has been tested. Therefore, it is theoretically possible, even if statistically unlikely, that there exist conflictual scenarios not covered by the simulations. This subsection and the next one presents two methods to deal with this issue. This subsection introduces **covering tests**. The main idea is to have a more accurate view on which scenarios are tested.

For instance, a request for Route `R_CC_102` can be made when none or several routes are already set in the station. Furthermore, a same route can have different states depending on which of its elements are released. Similarly to software testing where code coverage [91] is used to gain confidence into the quality of test suites, measures and report statistics related to the scenarios coverage for simulations can also be performed. More exactly, we record for each request the number of times it is generated and granted for different situations in the station.

The idea behind this test coverage is twofold. First, it aims to verify that the requests can be done in many different situations and secondly,

it can be used to detect conflictual routes. Table 4.4 summarizes this test coverage for the scenarios where an activation request is done when Route R_CXC_104 is already activated. After 1 hour of simulation, 361 496 requests were done under this assumption. Furthermore, each scenario occurred with a uniformly probability with a mean of 11661 and a standard deviation of 180.

Route	Ratio (%)	Route	Ratio	Route	Ratio
R_DXC_092	0	R_EC_091	0	R_CXC_102	0
R_KXC_103	22.65	R_CC_102	0	R_JC_012	14.75
R_CC_103	0	R_CGC_012	13.70	R_JXC_012	16.33
R_KC_101	24.37	R_CC_104	0	R_DXC_091	0
R_CC_101	24.85	R_KXC_101	24.13	R_CGC_011	13.69
R_KC_102	24.36	R_IC_011	14.51	R_CXC_103	0
R_EC_092	0	R_FC_091	0	R_KXC_102	24.15
R_KXC_104	21.89	R_KC_104	23.11	R_DC_092	12.57
R_DC_091	12.62	R_KC_103	22.78	R_IC_012	14.05
R_JXC_011	16.16	R_CXC_101	12.71	R_FC_092	0
R_JC_011	15.38	-	-	-	-

Table 4.4. Test coverage ($\frac{\#\text{Granted}}{\#\text{Done}}\%$) when route R_CXC_104 is activated for Braine l'Alleud.

This table summarises the proportion of times that a route activation request is granted after being issued. We can observe that some requests, like for R_DXC_092, are always refused when R_CXC_104 is activated. In Figure 2.1 we can indeed notice that R_DXC_092 is highly interleaved with R_CXC_104 such that there exists no state of R_CXC_104 where R_DXC_092 can be also activated.

For the other requests, they are all of them much less often granted than they are done. It is because other routes can also be set in the station, which will prevent the acceptance of the request. However, we can notice that some routes have a lower probability to be set than other. The requests having the lowest probability to be granted are for R_CXC_101, R_DC_091 and R_DC_092 which are all three interleaved with R_CXC_104. Generally speaking, the more a route is constrained, the

lower will be its probability to be activated.

Similar results are observed for scenarios involving other routes, or for other stations, which shows that most of the scenarios are covered by the simulation.

4.6.6 Importance splitting

Following the same idea, **importance splitting** [92, 85] is a technique initially used for guiding the simulation to a particular state. Such particular states are called **rare events**. For instance, an accident can be considered as a rare event. In order to spot them, we can guide the simulation in order to bring it closer to an accident and finally try to produce it. If the accident can happen, it means that the interlocking is incorrect.

Importance splitting can then be used to increase the probability of generating rare events but also to speed up the errors detection by decreasing the number of simulations required to estimate the probability.

It starts by splitting the rare event in a sequence of temporal properties φ_k, with the logical characteristic:

$$\varphi = \varphi_M \Rightarrow \varphi_{M-1} \Rightarrow \cdots \Rightarrow \varphi_1.$$

This defines a set of levels, each associated to the conditional probability

$$Pr(\rho \models \varphi_{k+1} | \rho \models \varphi_k)$$

of reaching level $k+1$ from level k. Instead of trying to verify directly the rare property, the importance splitting algorithm considers a set of sub-properties easier to verify and which lead progressively to the final property. The goal of simulations turns thereby to reach iteratively sub-properties until the rare event property is reached. An illustration of this process is presented on Figure 4.5 for three levels and with five simulations per level.

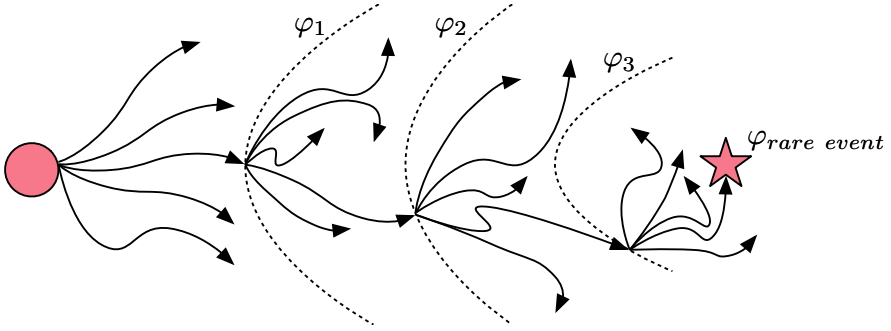


Figure 4.5. Illustration of importance splitting Algorithm for a budget of 5 simulations for 3 levels.

This process is presented in Algorithm 4.1. At Level k, the algorithm selects the prefix of the Traces $\tilde{\rho}_j^k$ that have reached Level $k-1$, and it continues these traces until they satisfy Property φ_k, or reach a stop condition (line 4). The algorithm computes the conditional probability of the level as the ratio of successful traces (line 5-6). Finally, successful traces are kept for the next level (line 7), and the unsuccessful ones are replaced by selecting randomly a successful trace (line 8). If the levels are adequately defined, the conditional probabilities can be estimated with a small simulation budget, and the importance splitting algorithm computes the probability of the rare property as the product of these conditional probabilities (line 9). The algorithm estimates iteratively these conditional probabilities with a fixed number of simulations (N) for each level. Importance splitting can then also be used in order to obtain more confidence on the error detection of the simulations.

Algorithm 4.1: Importance splitting

1 Let *stop* be a termination condition
2 $\forall j \in \{1,\dots,N\}$, set prefix $\tilde{\rho}_j^1 = \epsilon$ (empty path)
3 **for** $1 \leq k \leq M$ **do**
4 $\quad \forall j \in \{1,\dots,N\}$, using prefix $\tilde{\rho}_j^k$, generate path ρ_j^k until $(\rho_j^k \models \varphi_k) \vee stop$
5 $\quad I_k = \{j | j \in \{1,\dots,N\} \wedge \rho_j^k \models \varphi_k\}$
6 $\quad \tilde{\gamma}_k = \frac{|I_k|}{N}$
7 $\quad \forall j \in I_k$, $\tilde{\rho}_j^{k+1} = \rho_j^k$
8 $\quad \forall j \notin I_k$, let $\tilde{\rho}_j^{k+1}$ be a copy of ρ_i^k with $i \in I_k$ chosen uniformly randomly
9 $\tilde{\gamma} = \prod_{k=1}^{M} \tilde{\gamma}_k$

Furthermore, Equation (4.8) shows how a $(1-\alpha)$ confidence interval can be obtained [85]:

$$\left[\tilde{\gamma}\left(\frac{1}{1+\frac{z_\alpha \sigma}{\sqrt{N}}}\right), \tilde{\gamma}\left(\frac{1}{1-\frac{z_\alpha \sigma}{\sqrt{N}}}\right)\right] \tag{4.8}$$

where z_α is the $(1-\alpha/2)$ quantile of the standard normal distribution and σ the standard deviation estimated by $\sum_{k=1} M \frac{1-\tilde{\gamma}_k}{\tilde{\gamma}_k}$.

4.6.7 Experiments

⟦Results must be updated for the new model⟧

The aim of this subsection is to analyse the validity of this approach through experimental results. Indeed, now that we have a model and a verification process, we need to ensure that it will efficiently detect the errors leading to safety or availability issues. As stated previously, these experiments have been carried out with a MacBook Pro 2.6 GHz Intel Core i5 processor and with a RAM of 16 Go 1600 MHz DDR3 using a 64-Bit HotSpot(TM) JVM 1.8 on Yosemite 10.10.5.

Concerning the statistical model checking algorithms, we use the implementation and the framework proposed by PLASMA lab platform [93, 94]. PLASMA Lab is a compact, efficient and flexible platform for statistical model checking of stochastic models. It includes several statistical model checking algorithms and a library to include new simulators and to define properties. Simulators of systems or models can be reused with few implementation work thanks to the existing libraries.

It already includes simulators for Reactive Module Language (Markov chains models as in the PRISM model-checker [95]), Simulink [96] and SystemC [97] models. In our case, we have developed a new plug-in that implements PLASMA Lab library and creates an interface between PLASMA Lab and the simulator presented in Section 3.7.

The rest of this section presents the experiments that have been carried out in order to analyse the reliability of our method.

Error detection The first experiment consists in introducing several errors in the application data in order to test if they are detected through simulations. The errors introduced are described in Section 4.3. Using Monte Carlo estimations, we performed one hundred simulations of one day. Table 4.5 recaps the execution time and the probability to detect issues violating the requirements formalised on Section 4.2 with a single 1-day simulation when errors are introduced in the application data of Braine l'Alleud (Figure 2.1). Simulating the trace and verifying all the requirements for a single simulation take approximately [[XX]] seconds.

	Safety requirements			Availability requirements	Time
	(1)	(2)	(3)	(4)	
a.	0	100	0	0	1424
b.	0	0	100	100	1086
c.	91	69	29	63	1348
d.	93	100	0	33	1845
e.	72	97	0	79	1199
f.	0	0	0	100	1652

Table 4.5. Execution time (in seconds) and probability (in percent) of detecting an issue violating requirements of Section 4.2 with a single 1-day simulation when errors are inserted on Braine l'Alleud application data.

A non zero probability means that a safety or an availability issue has occurred on at least one simulation. In this case, we have then the certainty that the interlocking is incorrect. As we can see on this table, each error introduced in the application data causes the violation of at least one requirement which means that all the errors have been detected through 100 simulations of one day. Using these results, it is possible to analyse which issues are caused by specific errors. For instance, injecting an error of type f. only causes availability issues.

Importance splitting for collision detection The last experiment deals with `no collision` Requirement. As explained in the previous subsection, importance splitting can be used to speed up the errors detection. Furthermore, `no collision` Requirement can easily be split in different levels. It seems then suitable to verify this requirement using

Statistics	Importance splitting	Monte Carlo
# experiments	10	1
# simulations per experiment	100	1000
Average execution time (sec)	257	1320
Average arithmetic mean (%)	93.9	93.94
Standard deviation (%)	1.21	0.75
99.9%-confidence interval (%)	$[92.71, 95.12]$	$[91.43, 96.45]$

Table 4.6. Comparison between Monte Carlo and importance splitting algorithms for collision detection when an error of type f. is introduced on Braine l'Alleud application data

importance splitting.

The first step is to define the different levels. The first level is reached when two conflictual routes are set together in the station. Two routes are conflictual if they share at least one common track segment. For instance, `R_CXC_102` and `R_DXC_091` on Figure 2.1 are conflictual. The next level is reached when there is only one track segment between two trains following conflictual routes if and only if no train is beyond the track segment where the collision can occur. According to the previous example, if the train following Route `R_CXC_102` is on Track segment `T_01BC` and the train following Route `R_DXC_091` is on Track `102`, there is only a difference of one track segment. The third level is the event that we want to detect: the collision. According to importance splitting algorithm, simulations reaching a level are recorded and then used as a new start point for next simulations.

Once the levels are determined, importance splitting can be used. Table 4.6 presents statistics for simulations of one day with an error of type d. on the application data. The number of experiments and the number of simulations per experiment are chosen in order to have the same total number of simulations for Monte Carlo estimation and importance splitting (1000 in that case). As we can see, even for a medium size station such as Braine l'Alleud where errors are rapidly detected with Monte Carlo, importance splitting can give similar results much faster for a same number of simulations.

4.7 Dedicated algorithm

4.7.1 Motivation

Section 4.6 presented an innovative approach for the verification of railway interlocking systems. By taking the advantages of model checking and random simulation, a parametrizable confidence on the reliability and the availability of the entire system can be obtained.

This solution can be categorised as tests performed on a probabilistic model. However, although the last version of CENELEC EN50128 Standard [2] describes probabilistic tests, they are not considered as approved verification tests for the safety. Even, if this approach can also be referenced as black box tests [98] which are accepted by the standard, its industrialisation is compromised. It is why it is preferable to consider exhaustive methods for the safety requirements. Concerning availability requirements, exhaustiveness is not mandatory and statistical model checking remains then suitable for them.

As presented in Section 4.4, model checking is the commonest approach for performing an exhaustive verification, but is also limited to the state space explosion problem. Several improvements have been provided in order to restrict it but all of them are generic and although they can be applied for any model checking applications, they do not take advantage of the intrinsic specificities of the considered system. Even if model checking uses knowledge of the system for its modelling, it does not take advantage of this knowledge for designing the verification algorithm. Our idea is to use our knowledge of the railway field in order to design the verification algorithm. The rest of this section describes the algorithm designed and analyses its performances through theoretical and experimental results.

4.7.2 Verification

The goal of the **dedicated algorithm** is to use specificities of the interlocking system in order 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. Roughly speaking, this approach is related to model checking. Indeed, an automatic and exhaustive verification of a model is still performed, but now this verification is limited to a limited state space that increases slower than a complete model checking approach. The rest of this subsection describes the entire process of the algorithm and states the assumption under which it can be used.

Algorithm The first step is to initialise the variables. The inputs and the outputs of the algorithm are shown in Algorithm 4.2. 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.

Algorithm 4.2: Initialisation

1 **Input:** a railway station S
2 **Output:** **True** if S satisfies the requirements, **False** otherwise
3 Let $ROUTES$ be the set of all routes in S
4 Let $TRACK\_SEGMENTS$ be the set of all track segments in S
5 Let $POINTS$ be the set of all points in S
6 Let $COMPONENTS$ be the set of all physical components in S

The idea behind this algorithm is to verify that no issue occurs in any situations, 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 also related to the **monotonicity** of the application data.

Proposition 4.2. *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 2.1) 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. However, this property can be easily checked through a static analysis. To do so, one can simply read sequentially the application data and check separately each condition. □

Proposition 4.3. *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 so 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 $C_n \subseteq C$, the set of components used or locked by Route r_n. 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 issues can be represented as intersections between such sets. An intersection is formed by at least two routes. Two routes are then sufficient to detect any safety issues provided that commanding a third route will not relax C_1 or C_2 by releasing some components thereafter. According to Proposition 4.2, 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 [99] where the verification is limited to two trains. Algorithm 4.3 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 2.1 and 2.4. The bidirectional locking request (Listing 2.3) 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 2.5. 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 `no collision` 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 `no derailment1` is tested each time r_2 has been commanded. If the current position of t_1 is a point, the point cannot

Algorithm 4.3: 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
```

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 `no derailment2` is tested on lines 24 and 25 as a sub requirement stating that trains must reach their destination. If r_1 cannot be activated (lines 8-9), we can consider that we have an availability issue because no other route is already activated.

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**, still on a MacBook Pro 2.6 GHz Intel Core i5 processor and with a RAM of 16 Go 1600 MHz DDR3 using a 64-Bit HotSpot(TM) JVM 1.8 on Yosemite 10.10.5.

4.7.3 Experiments

Several kinds of errors related to safety (all of the errors presented in Section 4.3, except (e) which is related to availability) have been introduced in the application data of Braine l'Alleud (Figure 2.1) in order to test the adequacy of the dedicated algorithm. All of them have been successfully detected.

Furthermore, we perform three experimentations in order to analyse the scalability of the algorithm. Firstly, we compare the execution time required to verify different numbers of routes in the station. 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. [5] that have performed a verification of Nameche with model checking. Figure 4.6 recaps the execution time of the different experimentations.

Notice that the 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. Indeed, the verification is performed in less than 3 minutes for Braine l'Alleud and in less than 16 minutes for Courtrai.

These experimental results can then give more confidence about the reliability and the scalability of this dedicated algorithm.

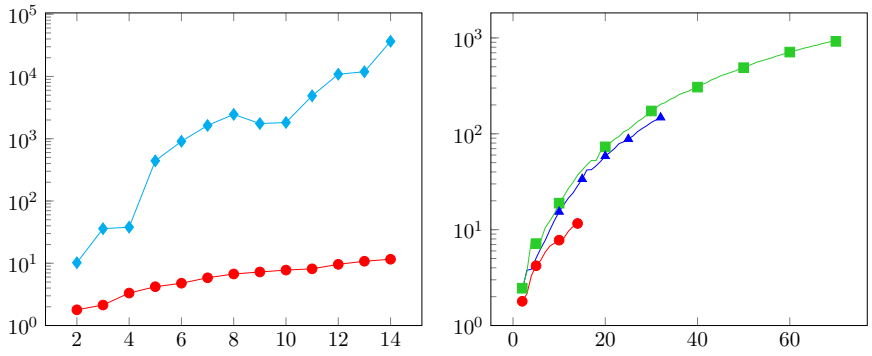


Figure 4.6. Execution time (in seconds) in function of the number of routes in Nameche (●), Braine l'Alleud (■) and Courtrai (▲) by using our algorithm and the model checking approach of Busard et al. [5] for Nameche (⧫).

4.8 Related work

Many other works related to interlocking verification exist in the literature, each with their own specificities. This section recaps some of them:

- Hartonas-Garmhausen et al. [74] propose a verification based on real time constraints. They use Verus Language [100] for the modelling and express the properties as invariants in Computational Tree Logic (CTL).
- Haxthausen et al. [101] use bounded model checking for the verification in order to deal with the state space explosion problem.
- Haxthausen et al. [102] detail how they modelled an European Train Control System (ETCS) level 2 compatible Danish interlocking. The state space, the transition relations and the safety properties are efficiently evaluated by solvers based on Satisfiability modulo theory (SMT) [103] that support bit vector and integer arithmetic. They also model the sequential release feature.
- Many works [104, 105, 106] use Petri net for the modelling.
- Moler et al. propose in several works [99, 107, 108] to use CSP||B Language for the verification [109]. The overall specification com-

bines two communicating models, one made of CSP process descriptions and one made of a collection of B machines.

- Abo et al. [110] explains how OVADO Tool can be used in order to perform the verification with B Language [111].
- More generally, Fantechi et al. [112] give an overview of the commonest methods for interlocking verification.

However, best of our knowledge, there exists no work dealing with our specific case: the automatic and generic verification of route based interlockings expressed in SSI and scalable for any station. Furthermore, the methods introduced in our work have never been used yet for the verification of any interlocking systems.

Chapter 5
Verification toolbox

In this section, we will detail the software implemented, its characteristics, its functionalities and an user manual.

Chapter 6
Conclusion

The research goal of this thesis is to design and develop innovative methods in order to perform an automatic verification of railway interlocking systems. Chapter 2 introduced the principles of interlocking systems. Chapter 3 explained how such a system could be modelled while Chapter 4 proposed different methods in order to carry out the verification of the model. Finally Chapter 5 presented the software implemented during this thesis. This chapter recaps first the contributions provided in this thesis and then sketches the future perspectives.

6.1 Contribution of this thesis

This section recaps our current contributions:

- A tool designed to automatically parse application data on SSI format. The output obtained can be used for several purposes and different kinds of modelling. It has been presented in Section 3.2.
- A parser tool extracting the topology of a station from a data source based on railML. It has been presented in Section 3.3.
- A model instantiating the behaviour of a route based interlocking system from its application data and the topology of its station. The model presented here is designed in order to allow its simulation by a discrete event simulator and its verification with different approaches. The model has been presented in Chapter 3.
- The formalisation of the safety properties of [5] in BLTL. Such logic is used in order to have the possibility to determine when the simulator must stop its processing. The formalisation has been presented in Section 4.2.

- The introduction of an availability property that an interlocking must face in order to ensure that no train would be stuck in a station. Such a property has also been formalised in BLTL. It has also been formalised in Section 4.2.
- A DES engine which can be run on the top of the previous model. This tool contains several advanced features such as the possibility to stop a simulation at any state and to replay it later. It has been presented in Section 3.7.
- A new verification method based on a random simulation which is executed on the top of the DES engine. It is presented in section 4.5.
- The utilisation of SMC methods and principles, such as Monte Carlo estimation, Chernoff's bound, covering tests and importance splitting algorithm, for improving the verification by random simulation and obtaining then more confidence on the reliability of the system. It has been presented in Section 4.6.
- A polynomial dedicated algorithm verifying that an interlocking will never cause derailments or collisions provided that an assumption of monotonicity hold. it can also verify that each train will reach the correct destination. It has been presented in Section 4.7.
- An executable tool instantiating the model, its simulation, the properties and the different verification methods. A command line interface as well as a graphical user interface have also be developed in order to facilitate its utilisation and its ergonomics.

6.2 Perspectives

The work presented here is currently in progress. Our next steps will probably be the following:

- For now, the experimental results were obtained from the analysis of Namêche and Braine l'Alleud. The station of Courtrai is still on the modelling process. Once modelled, we will perform experimentations on it. The analysis of Courtrai will probably close the verification chapter.

- Optimising and refining the implemented software for its future utilisation.
- Beyond the verification, we will see how optimisation of interlocking system can be performed.

    1. Discuss with experts on what are the interesting cases and how optimisation can help.
    2. Review the state of the art optimisation methods related to this field.
    3. See where are the lacks and fill them with methods never used yet for interlocking optimisation.

Bibliography

[1] G. Theeg, E. Anders, and S. Vlasenko, *Railway Signalling & Interlocking: International Compendium.* Eurailpress, 2009.

[2] E. CENELEC, "50128," *Railway applications-Communication, Signaling and Processing Systems-Software for Railway Control and Protection Systems*, 2011.

[3] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification*, pp. 1–30, Springer, 2012.

[4] A. Cribbens, "Solid-state interlocking (ssi): an integrated electronic signalling system for mainline railways," in *IEE Proceedings B (Electric Power Applications)*, vol. 134, pp. 148–158, IET, 1987.

[5] S. Busard, Q. Cappart, C. Limbrée, C. Pecheur, and P. Schaus, "Verification of railway interlocking systems," in *Proceedings 4th International Workshop on Engineering Safety and Security Systems, ESSS 2015, Oslo, Norway, June 22, 2015.*, pp. 19–31, 2015.

[6] C. Limbrée, Q. Cappart, C. Pecheur, and S. Tonetta, "Verification of railway interlocking, compositional approach with ocra," in *International Conference Reliability, Safety and Security of Railway Systems: Modelling, Analysis, Verification and Certification [TO UPDATE].*

[7] A. Nash, D. Huerlimann, J. Schütte, and V. P. Krauss, "Railml–a standard data interface for railroad applications," *Computers in Railways IX, WIT Press, Southampton*, pp. 233–240, 2004.

[8] Q. Cappart, C. Limbrée, P. Schaus, and A. Legay, "Verification by discrete simulation of interlocking systems," in *Proceedings of the 29th Annual European Simulation and Modelling Conference*, EUROSIS, October 2015.

[9] N. C. EN, "50129: Railway application–communications, signaling and processing systems–safety related electronic systems for signaling," *British Standards*, 2003.

[10] B. Livshits, *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford University, 2006.

[11] R. S. Scowen, "Extended bnf-a generic base standard," tech. rep., Technical report, ISO/IEC 14977. http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf, 1998.

[12] J. H. Earle, D. Olsen, *et al.*, *Engineering Design Graphics: AutoCAD Release 14*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[13] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*. Artima Inc, 2008.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, *et al.*, *Object-oriented modeling and design*, vol. 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[15] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–14, ACM, 1992.

[16] K. Arnold, J. Gosling, D. Holmes, and D. Holmes, *The Java programming language*, vol. 2. Addison-wesley Reading, 2000.

[17] N. Chomsky, "Three models for the description of language," *Information Theory, IRE Transactions on*, vol. 2, no. 3, pp. 113–124, 1956.

[18] S. Ginsburg, *The Mathematical Theory of Context Free Languages.[Mit Fig.]*. McGraw-Hill Book Company, 1966.

[19] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[20] M. Lange and H. Leiß, "To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm," *Informatica Didactica*, vol. 8, pp. 2008–2010, 2009.

[21] B. Ford, "Packrat parsing:: simple, powerful, lazy, linear time, functional pearl," in *ACM SIGPLAN Notices*, vol. 37, pp. 36–47, ACM, 2002.

[22] A. J. D. Reis, "Recursive-descent parsing," *Compiler Construction Using Java, JavaCC, and Yacc*, pp. 185–214.

[23] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *ACM Sigplan Notices*, vol. 21, pp. 38–45, ACM, 1986.

[24] G. Hutton, "Higher-order functions for parsing," *J. Funct. Program.*, vol. 2, no. 3, pp. 323–343, 1992.

[25] A. Moors, F. Piessens, and M. Odersky, "Parser combinators in scala," 2008.

[26] T. Lam, J. J. Ding, J.-C. Liu, *et al.*, "Xml document parsing: Operational and performance characteristics," *Computer*, no. 9, pp. 30–37, 2008.

[27] W. S. Means and M. A. Bodie, "The book of sax," 2002.

[28] D. B. West *et al.*, *Introduction to graph theory*, vol. 2. Prentice hall Upper Saddle River, 2001.

[29] S. H. Strogatz, "Exploring complex networks," *Nature*, vol. 410, no. 6825, pp. 268–276, 2001.

[30] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, pp. 251–262, ACM, 1999.

[31] S. P. Borgatti, "2-mode concepts in social network analysis," *Encyclopedia of complexity and system science*, vol. 6, 2009.

[32] R. C. Thomson and D. E. Richardson, "A graph theory approach to road network generalisation," in *Proceeding of the 17th international cartographic conference*, pp. 1871–1880, 1995.

[33] W. Wong, "A simple graph theory and its application in railway signaling," in *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the*, pp. 395–409, IEEE, 1991.

[34] J. O'Madadhain, D. Fisher, S. White, and Y. Boey, "The jung (java universal network/graph) framework," *University of California, Irvine, California*, 2003.

[35] 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*, pp. 101–107, Australian Computer Society, Inc., 2006.

[36] D. Tombs, N. Robinson, G. Nikandros, *et al.*, "Signalling control table generation and verification," *CORE 2002: Cost Efficient Railways through Engineering*, p. 415, 2002.

[37] M. F. Jentsch, A. S. Bahaj, and P. A. James, "Climate change future proofing of buildings - generation and assessment of building simulation weather files," *Energy and Buildings*, vol. 40, no. 12, pp. 2148–2168, 2008.

[38] M. D. Rossetti and S. Nangia, "An object-oriented framework for simulating full truckload transportation networks," in *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pp. 1869–1877, IEEE Press, 2007.

[39] M. M. Fioroni, L. A. G. Franzese, I. R. de Santana, P. E. P. Lelis, C. B. da Silva, G. D. Telles, J. A. S. Quintáns, F. K. Maeda, and R. Varani, "From farm to port: simulation of the grain logistics in brazil," in *Proceedings of the 2015 Winter Simulation Conference*, pp. 1936–1947, IEEE Press, 2015.

[40] T. Worth, R. Uzsoy, E. Samoff, A.-M. Meyer, J.-M. Maillard, and A. M. Wendelboe, "Modelling the response of a public health

department to infectious disease," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pp. 2185–2198, IEEE, 2010.

[41] X. Jin, A. I. Sivakumar, and S. Y. Lim, "A simulation based analysis on reducing patient waiting time for consultation in an outpatient eye clinic," in *Simulation Conference (WSC), 2013 Winter*, pp. 2192–2203, IEEE, 2013.

[42] S. Sogin, C. P. Barkan, and M. R. Saat, "Simulating the effects of higher speed passenger trains in single track freight networks," in *Proceedings of the Winter Simulation Conference*, pp. 3684–3692, Winter Simulation Conference, 2011.

[43] A. Nash and D. Huerlimann, "Railroad simulation using opentrack," *Computers in railways IX*, pp. 45–54, 2004.

[44] S. Robinson, *Simulation: the practice of model development and use*. Palgrave Macmillan, 2014.

[45] P. Checkland, "Systems thinking, systems practice," 1981.

[46] A. Sulistio, C. S. Yeo, and R. Buyya, "A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools," *Software-Practice and Experience*, vol. 34, no. 7, pp. 653–674, 2004.

[47] G. Fishman, *Discrete-event simulation: modeling, programming, and analysis*. Springer Science & Business Media, 2013.

[48] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Mathematical Systems Theory*, vol. 10, no. 1, pp. 99–127, 1976.

[49] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[50] J. E. Gentle, *Random number generation and Monte Carlo methods*. Springer Science & Business Media, 2006.

[51] H. Niederreiter, "Quasi-monte carlo methods and pseudo-random numbers," *Bulletin of the American Mathematical Society*, vol. 84, no. 6, pp. 957–1041, 1978.

[52] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering*. John Wiley & Sons, 2010.

[53] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[54] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, vol. 15, no. 2, pp. 364–383, 1986.

[55] B. A. Wichmann and I. D. Hill, "Algorithm as 183: An efficient and portable pseudo-random number generator," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 31, no. 2, pp. 188–190, 1982.

[56] J. Eichenauer and J. Lehn, "A non-linear congruential pseudo random number generator," *Statistische Hefte*, vol. 27, no. 1, pp. 315–326, 1986.

[57] D. Knuth, "The art of computer programming, volume two, seminumerical algorithms," 1998.

[58] J. R. Kelly, "Cryptographically secure pseudo random number generator," Aug. 14 2001. US Patent 6,275,586.

[59] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[60] OscaR Team, "OscaR: Scala in OR," 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[61] K. Müller and T. Vignaux, "SimPy: Simulating Systems in Python," *ONLamp.com Python DevCenter*, February 2003.

[62] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[63] N. Kamide, "Bounded linear-time temporal logic: A proof-theoretic investigation," *Annals of Pure and Applied Logic*, vol. 163, no. 4, pp. 439–466, 2012.

[64] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[65] K. Winter, "Model checking railway interlocking systems," in *Australian Computer Science Communications*, vol. 24, pp. 303–310, Australian Computer Society, Inc., 2002.

[66] K. Winter and N. J. Robinson, "Modelling large railway interlockings and model checking small ones," in *Proceedings of the 26th Australasian computer science conference-Volume 16*, pp. 309–316, Australian Computer Society, Inc., 2003.

[67] K. Winter, "Optimising ordering strategies for symbolic model checking of railway interlockings," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pp. 246–260, Springer, 2012.

[68] A. Mirabadi and M. Yazdi, "Automatic generation and verification of railway interlocking control tables using fsm and nusmv," *Transport Problems*, vol. 4, no. 1, pp. 103–110, 2009.

[69] K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[70] M. Huber and S. King, "Towards an integrated model checker for railway signalling data," in *FME 2002: Formal Methods - Getting IT Right*, pp. 204–223, Springer, 2002.

[71] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification*, pp. 1–30, Springer, 2011.

[72] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi, "Model checking interlocking control tables," in *FORMS/FORMAT 2010*, pp. 107–115, Springer, 2011.

[73] C. Eisner, "Using symbolic model checking to verify the railway stations of hoorn-kersenboogerd and heerhugowaard," in *Correct*

Hardware Design and Verification Methods, pp. 99–109, Springer, 1999.

[74] V. Hartonas-Garmhausen, S. Campos, A. Cimatti, E. Clarke, and F. Giunchiglia, "Verification of a safety-critical railway interlocking system with real-time constraints," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pp. 458–463, IEEE, 1998.

[75] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[76] S. Busard and C. Pecheur, "Pynusmv: Nusmv as a python library," in *Nasa Formal Methods*, pp. 453–458, Springer, 2013.

[77] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*, pp. 122–135, Springer, 2010.

[78] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *Computer Aided Verification*, pp. 266–280, Springer, 2005.

[79] S. K. Jha, E. M. Clarke, C. J. Langmead, A. Legay, A. Platzer, and P. Zuliani, "A bayesian approach to model checking biological systems," in *Computational Methods in Systems Biology*, pp. 218–234, Springer, 2009.

[80] E. M. Clarke, J. R. Faeder, C. J. Langmead, L. A. Harris, S. K. Jha, and A. Legay, "Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway," in *Computational Methods in Systems Biology*, pp. 231–250, Springer, 2008.

[81] E. Clarke, A. Donzé, and A. Legay, "On simulation-based probabilistic model checking of mixed-analog circuits," *Formal Methods in System Design*, vol. 36, no. 2, pp. 97–113, 2010.

[82] A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay, "Statistical abstraction and model-checking of large heterogeneous systems," in *Formal Techniques for Distributed Systems*, pp. 32–46, Springer, 2010.

[83] R. Grosu and S. A. Smolka, "Monte carlo model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 271–286, Springer, 2005.

[84] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method*, vol. 707. John Wiley & Sons, 2011.

[85] C. Jegourel, A. Legay, and S. Sedwards, "An effective heuristic for adaptive importance splitting in statistical model checking," in *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pp. 143–159, Springer, 2014.

[86] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *The Annals of Mathematical Statistics*, pp. 493–507, 1952.

[87] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[88] J. Barnat, L. Brim, and P. Rockai, "Scalable multi-core ltl model-checking," in *SPIN*, vol. 7, pp. 187–203, Springer, 2007.

[89] G. Holzmann, "The design of a distributed model checking algorithm for spin," *FMCAD, Invited Talk*, 2006.

[90] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *Cloud computing*, pp. 115–131, Springer, 2009.

[91] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[92] C. Jegourel, A. Legay, and S. Sedwards, "Importance splitting for statistical model checking rare properties," in *Computer Aided Verification*, pp. 576–591, Springer, 2013.

[93] B. Boyer, K. Corre, A. Legay, and S. Sedwards, "Plasma-lab: A flexible, distributable statistical model checking library," in *Quantitative Evaluation of Systems*, pp. 160–164, Springer, 2013.

[94] C. Jegourel, A. Legay, and S. Sedwards, "A platform for high performance statistical model checking–plasma," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 498–503, Springer, 2012.

[95] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "Prism: A tool for automatic verification of probabilistic systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 441–444, Springer, 2006.

[96] S. T. Karris, "Introduction to simulink with engineering applications," 2006.

[97] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*, vol. 71. Springer Science & Business Media, 2009.

[98] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[99] F. Moller, H. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Defining and Model Checking Abstractions of Complex Railway Models Using CSP‖B," in *Hardware and Software: Verification and Testing* (A. Biere, A. Nahir, and T. Vos, eds.), vol. 7857 of *Lecture Notes in Computer Science*, pp. 193–208, Springer Berlin Heidelberg, 2013.

[100] S. Campos and E. Clarke, "The verus language: representing time efficiently with bdds," in *Transformation-Based Reactive Systems Development*, pp. 64–78, Springer, 1997.

[101] A. E. Haxthausen, J. Peleska, and R. Pinger, "Applied bounded model checking for interlocking system designs," in *Software Engineering and Formal Methods*, pp. 205–220, Springer, 2013.

[102] L. H. Vu, A. E. Haxthausen, and J. Peleska, "Formal modeling and verification of interlocking systems featuring sequential release,"

in *Formal Techniques for Safety-Critical Systems*, pp. 223–238, Springer, 2014.

[103] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories.," *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[104] S. Vanit-Anunchai, "Modelling railway interlocking tables using coloured petri nets," in *Coordination Models and Languages*, pp. 137–151, Springer, 2010.

[105] M. Antoni and N. Ammad, "Formal validation method and tools for french computerized railway interlocking systems," in *Railway Condition Monitoring, 2008 4th IET International Conference on*, pp. 1–10, IET, 2008.

[106] P. Sun, S. Collart-dutilleul, and P. Bon, "A model pattern of railway interlocking system by petri nets," in *Models and Technologies for Intelligent Transportation Systems (MT-ITS), 2015 International Conference on*, pp. 442–449, IEEE, 2015.

[107] F. Moler, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Combining event-based and state-based modelling for railway verification," 2012.

[108] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Cspb modelling for railway verification: the double junction case study," in *Proceedings of the 12th International Workshop on Automated Verification of Critical Systems*, 2012.

[109] M. Butler and M. Leuschel, "Combining csp and b for specification and property verification," in *FM 2005: Formal Methods*, pp. 221–236, Springer, 2005.

[110] R. Abo and L. Voisin, "Data formal validation of railway safety-related systems: Implementing the ovado tool," *Towards a Formal Methods Body of Knowledge for Railway Control and Safety Systems*, p. 27, 2013.

[111] K. Lano, *The B language and method: a guide to practical formal development*. Springer Science & Business Media, 2012.

[112] A. Fantechi, W. Fokkink, and A. Morzenti, "Some trends in formal methods applications to railway signaling," *Formal Methods for Industrial Critical Systems: A Survey of Applications*, pp. 61–84, 2013.