VERIFICATION BY DISCRETE SIMULATION OF INTERLOCKING SYSTEMS

Quentin Cappart
Christophe Limbrée
Pierre Schaus
Université catholique de Louvain
Place de l'Université 1, 1348, Belgium
E-mail: quentin.cappart@uclouvain.be

Axel Legay
INRIA/IRISA, Rennes, France

KEYWORDS

Discrete event simulation, Verification, Railway, Interlocking

ABSTRACT

In the railway domain, an interlocking is the system controlling active components in a station in order to ensure a safe train traffic. The behaviour of modern interlockings is defined by particular data, called application data, describing the actions that the interlocking can take and under which conditions. However, application data are either prepared manually or prepared automatically by tools that do not guarantee a sufficient level of safety. Given the high level of safety required by such a system, the verification of the application data is a critical concern. Recent researches dealing with this issue are based on model checking. Due to the state space explosion problem, this approach does unfortunately not scale for large stations. In this paper, we present an innovative approach for the verification of interlocking data, based on a discrete event simulation, which does not suffer of the state space explosion problem. Although sacrificing exhaustiveness, we show experimentally on a real life instances that this approach is able to detect any introduced errors in the application data within seconds.

INTRODUCTION

Each train station is controlled by a system having the responsibility to ensure safe movements of trains and to avoid all risk of conflicts between their path. Such a system is called an **interlocking**. More specifically, an interlocking controls the physical components of the infrastructure such as the points and the authorities of movement (e.g. signals) in order to safely allow the trains through a station.

Unlike the interlockings of the first generation that were based on a mechanical or relay logic, modern interlockings are computer based which means that the actions are calculated by a software. Furthermore, the software of computer-based interlockings relies on configuration data specifying what are the possible actions and under which conditions they can be taken (Theeg et al., 2009). Such data are called **application data** and are specific to each station.

The safety of the train traffic inside a station is thereby highly dependent on the correctness of its application data. However, the application data are prepared manually and are thus subject to human errors leading their verification to a critical concern. One approach for interlocking verification deeply studied in the literature is model checking (Huber and King, 2002; Winter et al., 2006; Busard et al., 2015). The goal of a model checker is to verify if a system meets a set of safety properties by considering all the reachable states of the model representing the system. This method is exhaustive, in other words, if a requirement is not satisfied, it will always be detected.

However, model checking suffers from the state explosion problem (Clarke et al., 2012). Whereas small sized stations can be verified efficiently the verification time grows exponentially as the size of the station increases and might not return a result within a reasonable time in practice. This is a well known limitation for model checking. Several techniques to limit this problem have been proposed. Winter and al. (Winter et al., 2006) propose to relax the verification by reducing the complexity of the model and to improve the verification process by using the properties of the system. In (Huber and King, 2002), Huber and King implemented a symbolic model checker with different optimisations like a dynamic variable re-ordering. Winter (Winter, 2012) also proposes several strategies to optimise the variable ordering. Eisner (Eisner, 1999) uses symbolic model checking. Busard and al. (Busard et al., 2015) proposed in their model a customized model-checking algorithms based on operation on the BDD.

Despite these optimisations, applying model checking on medium or large stations remains intractable. In (Busard et al., 2015) only very small stations could be verified within hours of computations. To overcome this issue, we propose in this paper a new approach for the verification based on discrete event simulation (Schriber et al., 2012) which does not suffer from the state space explosion problem but sacrificing the exhaustiveness property. We show experimentally on a medium size real life instances of a Belgian interlocking that this approach is able to detect any introduced errors in the application data within seconds.

Usage of simulation in the railway domain is not new. The company OpenTrack provides a railway simulation tool (Nash and Huerlimann, 2004) to verify the capacity of the railway network, the feasibility of the schedules, collect statistics about running times, etc. but which is not related to interlocking verification. Hon and Kollmann (Hon and Kollmann, 2006) proposes an hybrid model for the verification based on simulation coupled with model checking. For the simulation, they use the software Rhapsody (Gery et al., 2002) which takes test cases as input and check if a system is correct by simulating the test cases. However, the test cases must be elaborated manually and it turns thereby into a manual verifications which is different of what we want to do.

To the best of our knowledge there is no existing work considering a discrete event simulation for verifying an interlocking system. Our methodology can be used on any application data formats used in any country.

In this work we instantiate our approach to the SSI format (Cribbens, 1987) mainly used by Infrabel (`www.infrabel.be`) in Belgium. In the next section, we describe how an interlocking works on a real life interlocking instance. This case study corresponds to a medium sized railway interlocking system of a Belgian station. In Section 3, we explain the principles and the benefits of our discrete event simulation for the verification. Finally, in Section 5, we discuss the experimental results obtained on the case study.

INTERLOCKING PRINCIPLES

As previously said, the role of an interlocking is to prevent any conflicting movements while the trains move on their reserved routes in the station. This section explains how it is done in practice. To to so, let us first consider a case study, the station of Braine l'Alleud, which is a typical Belgian medium sized station. Figure 1 shows the track layout of Braine l'Alleud (all the variable names are not represented on the Figure).

On this figure, several elements can be identified:

- The **signals** (e.g. CC) are used to control the train traffic. Each route begins from a signal which can grant or prohibit access to the route for the trains.
- The **points** (e.g. P_01AC) are the movable components that allow trains to move from one track to another one. According to the Belgian convention, a point can be in a normal position (left) or in a reverse position (right).
- The **track segments** are portion of the track where a train can be detected. They are either occupied, or clear. Track segments are delimited each other by the **joints**.

The **physical components** are controlled and monitored by the interlocking. Besides, the interlocking software makes use of **logical components** :

- The **routes** are the paths that the trains follow when running through the station. Each route starts from a signal and finishes to another signal or to a track segment. In the application data, the routes have the following format: R_*src*_*dest* (e.g. R_CC_104) with *src* the name of the start signal and *dest* the name of the destination. A route can either be set if it is reserved for a train, or unset on the contrary.
- The **subroutes** are the contiguous segments that the trains follow inside a route. When a route is commanded for a train, a set of contiguous subroutes is locked establishing a path from the origin of the route to its destination. When not requested, subroutes are in a free state. In the application data, they have the following syntax: U_*src*_*dest* (e.g. U_KXC_20C).
- The immobilisation zones, also called **UIR**, are the components materialising the immobilisation (locking) of a set of points for a route. As for the subroutes, they can be on a locked or a free state depending on whether they are reserved for a route. If they are locked, the points attached to the UIR are not supposed to move. In the application data, they are presented like this: U_IR(*identifier*) with *identifier* the name of the UIR. Generally, the UIR identifier is related to the name of the points locked by it. For instance, U_IR(08BC) locks Point P_08BC.

Using these components, the interlocking can control the train traffic by commanding the routes. The actions that have to be done and the conditions under which they can be executed are described in the application data. To explain the content of the application data, let us consider the scenario where the route from the signal KC to the track 103 has to be set:

1. Firstly, the interlocking will verify whether the request for the route R_KC_103 can be granted.

```
*Q_R(KC_103)
   if    R_KC_103 xs,
         P_09C cfr, P_08AC cfr,
```

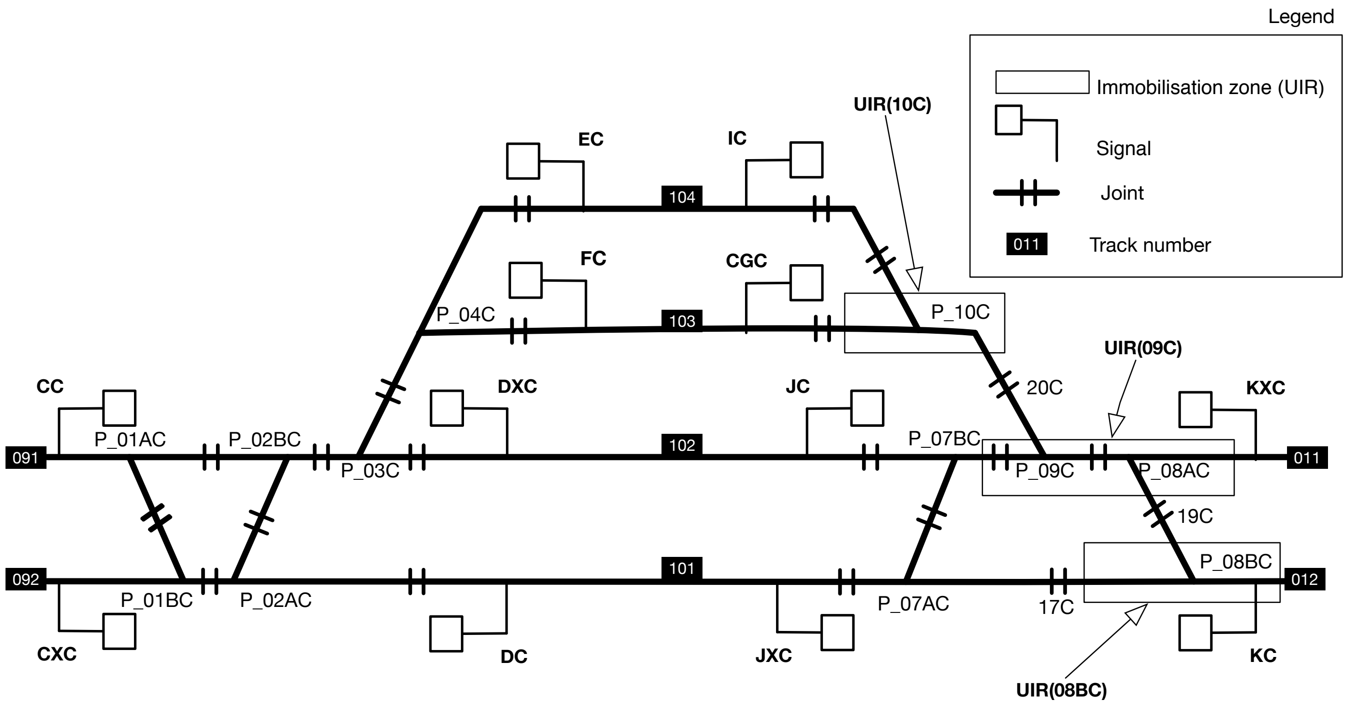

Figure 1: Layout of Braine l'Alleud Station

```
4             P_08BC cfr, P_10C cfn,
5             U_IR(09C) f, U_IR(08BC) f,
6             U_IR(10C) f
7      then   R_KC_103 s
8             P_09C cr, P_08AC cr,
9             P_08BC cr, P_10C cn,
10            U_IR(09C) l, U_IR(08BC) l,
11            U_IR(10C) l,
12            U_KC_19C l, U_19C_20C l,
13            U_20C_CGC l
```

Listing 1: Request for Setting Route R_KC_103

Listing 1 presents a typical route request as described in the SSI format. There is a similar request description for each route that can be commanded in the station. The first part of this request (line 2 to 6) are the conditions under which the request can be granted. More specifically, it can be granted if the route is not already set (xs value on line 2), if some points are free to be commanded to the reverse (cfr) or to the normal (cfn) position (line 3 and 4), and if some UIR are in a free state (f on line 5 and 6). The components requested can be seen on Figure 1.

2. Before moving a point, the interlocking must ensure that it can be moved without causing safety issues. Such conditions are also expressed in the application data.

```
*P_08BCN U_IR(08BC) f // normal position
*P_08BCR U_IR(08BC) f // reverse position
```

Listing 2: Conditions Allowing Point P_08BC to Move

Listing 2 states that the point P_08BC can be controlled in the normal position only if its UIR is free. The same condition is verified to control it in the reverse position. These conditions are the route setting conditions. If they are not satisfied, the route request is dropped.

3. If each condition is satisfied, the route can be set and the actions defined in line 7 to 13 of Listing 1 are taken. More precisely, the route is thoroughly set (s on line 7), the points are commanded either to reverse (cr) or to normal (cn) position (line 8 and 9), the UIR (line 10 and 11) and the subroutes (line 12 and 13) are locked. The home signal of the route (KC in our example) is controlled at its proceeding state when additional conditions are fulfilled. These additional conditions are abstracted in our model. At this step, the train can run through the station.

4. While the train is running through the route, reserved component can be progressively released. For route R_KC_103, when the train has cleared Track containing P_08BC, UIR(08BC) can be released. The conditions under which components can be released are also described in the application data.

```
U_KC_19C f
   if     R_KC_102 xs, R_KC_103 xs,
          R_KC_104 xs,
          T_08BC c
```

Listing 3: Subroute U_KC_19C release conditions

Listing 3 states that Subroute U_KC_19C can be released whether the routes on (line 2 and 3) are not set and the Track segment T_08BC (line 4) is free.

```
1 if U_IR(08BC) l then
2    if U_17C_KC f,
3       U_KC_17C f,
4       U_19C_KC f,
5       U_KC_19C f,
6    then U_IR(08BC) f
```

Listing 4: Conditions for Releasing UIR(08BC)

Listing 4 shows the release conditions for UIR(08BC): if it is locked (line 1), then it can be released (line 6) only if some subroutes are not locked (line 2 to 5). Unlike the reservation actions which are only executed upon request, the releasing actions are periodically verified.

This process briefly describes the route cycle controlled by the interlocking. To be more precise, real interlockings contains other components and other actions which are abstracted in our study. As previously said, verification of application data is a crucial task: an error or an omission can lead to serious safety issues. For instance, let us assume that the action `P_08BC cr` on Listing 1 is transformed on `P_08BC cn`. Following Figure 1, the train will move through Track T_07AC instead of Track T_08AC which can lead to a head to head collision from a train following Route R_JXC_012. There is thereby a real need of efficient and reliable methods to verify the application data.

VERIFICATION BY SIMULATION

On this section, we present a novel approach based on a **discrete event simulation** which does not suffer of the drawbacks of model checking. The idea is to simulate the train movements and the behaviour of an interlocking as described in its application data and to observe if any safety issues occurred. If no issue occurred and if 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 corresponding to cases that never occur in practice, the discrete simulation will only consider the cases which can potentially happen with a real interlocking.

A discrete event simulation involves three kinds of components:

- The **entities**, which are the active objects on which the simulation is applied. Each entity is characterized by its current state. Our model contains two families of entities: the interlocking components described in the previous section (e.g. the subroutes which can be free or locked), and the trains, characterized by their position.
- The **events**, which define actions that can alter the state of the entities and which can generate other events. On one hand, there are events for the actions defined in the application data: requesting a route and releasing a component. The execution of these actions is guarded by their conditions. The effect of these events is to set the considered entities into the requested state. On the other hand, there are the events triggered by the train movements: entering into the station, moving through it and leaving it.
- The **clock**, stating when the events must occur. Unlike a continuous simulation where event can occur during a time period, the discrete simulation requires each event to occur at a particular instant.

In our simulation, all of these components interact together as follows:

1. Trains randomly arrive in the station at the possible routes home signals. In practice, a train arrival is an event which can occur with a uniform probability on the discrete time interval $[t_a, t_a+n_a]$ where t_a is the time of the last train arrival ($t_a = 0$ for the first step) and n_a is a predefined parameter. Besides, each time such an event occurs, a new event is triggered in the interval $[t_a, t_a + n_a]$ while t_a is updated.

2. Route requests are periodically issued for the trains waiting at a start signal. If the route setting conditions are fulfilled, the route is set and all the actions described in the request are executed. Otherwise, the request is discarded and no action is taken. Like the trains arrival, a route request is an event which can occur in an interval $[t_r, t_r+n_r]$ with t_r the time of the last request and n_r a predefined parameter.

3. Trains move through the station following the path described by the station components state. Concretely, they always move forward from one track segment to the next one and follow the direction defined by the position of the points. The first movement of a Train x is triggered when its attached route request is accepted. The next movements occur in the interval $[t_m(x), t_m(x)+n_m]$ with $t_m(x)$ the time of the last movement done by Train x and n_m a parameter. Each train has thereby its own queue of events. By doing this, we implicitly model the fact that the speed of the trains can be different. The higher is n_m, the larger will be the speed difference between trains. Modelling an exact or a realistic speed has no importance for the verification. What matters is to have a n_m large

enough to allow the simulation to cover all the possible combinations of train positions. To do so, n_m must be higher than the largest number of movement steps for a train (i.e. the number of track segments composing the longest route). Moreover, the train lengths are abstracted, only the occupation of the track segments has an importance for the verification.

4. After each train movement, the system checks if a releasing event can be triggered. In this case, the requested components are thoroughly released.

5. When a train reaches the end of a route, it is removed from the station.

To model the randomness in our simulation we introduced parameters n for several kinds of events. Their goal is to define the time steps range on which the events can occur. Therefore, the values of n determines the frequency of occurrence of a family of events: the lower is n, the higher will be the frequency. Besides, the exact values of n has no importance, what matters is the relation between the values. A n lower than others indicates that the events related to n will have a higher probability to occur than the others. We choose to assign the same value than n_m for each n, which means that each event has the same probability to occur. As we will see in the next section, this default value provides good results. Figure 2 shows two possible scenarios for the event sampling for two trains with $n = 5$.

The model presented here is implemented in Scala using the discrete event simulation package of OscaR (OscaR Team, 2012). This toolkit has similar functionalities as SimPy (Müller and Vignaux, 2003). Once the simulation is launched, we can observe the expected behavior of the interlocking system as described by its application data and how it allows the trains to move through the station. The analysis of this behavior is finally used to verify the correctness of the application data. The key idea to perform the verification is to monitor the simulation to see whether a situation causing a safety issue has occurred. The monitoring can be done in two ways: with a dynamic GUI displaying the routes and the train movements in real time, or with the execution trace summarizing all the actions performed during the simulation. As identified in (Busard et al., 2015) there are three conflictual situations:

- Two trains are at the same time on the same track segment. In this situation, we have a collision between the two trains.
- A point is moving while the track segment containing it is occupied. This situation causes a derailment of the train.
- A train moves through a point not set in a position allowing the train to continue its path. For instance, in Figure 1, it occurs if a train follows Route R_CC_102 and if Point P_02BC is not set at the reverse position. It will also cause a derailment.

These situations can be expressed in terms of the state of the entities, and are easy to detect. Using this simulation, we can thereby verify in a non exhaustive way the correctness of the application data.

VALIDATION AND PERFORMANCE

In this section, we analyse the results obtained in order to validate the approach and its performance in terms of speed, test coverage and error detection capability.

The results presented here are based on two instances. A small sized station, Namêche, containing 7 points, 7 UIR, 7 signals, 14 routes and 26 subroutes. The other instance is our case study, Braine l'Alleud (Figure 1), which is a medium sized station containing 12 points, 10 UIR, 12 signals, 32 routes and 48 subroutes. Furthermore, we compared the results with the ones obtained using Busard et al. method (Busard et al., 2015) which made experiments on Namêche with a model checking approach.

In order to perform the validation, we introduced different kinds of errors leading to safety issues in the application data to check if they are thoroughly detected.

Table 1: Benchmark about Execution Time (in seconds) of Error Detection in seconds

	Namêche		Braine l'Alleud
	Simulation	Model checking	Simulation
1.	2.78 / 0.10	370	8.73 / 2.23
2.	2.94 / 0.45	242	6.88 / 2.57
3.	3.09 / 0.32	258	7.05 / 0.09
4.	3.32 / 0.19	163	6.71 / 1.28
5.	2.98 / 0.06	169	6.49 / 0.88

Table 1 recaps the execution time in seconds for detecting the following errors with the simulation and the model checking approach:

1. Missing condition in a route request.

2. Point moved to a wrong position when setting a route.

3. Subroute not properly locked when setting a route.

4. Condition missing for releasing a subroute.

5. Condition missing for releasing an UIR.

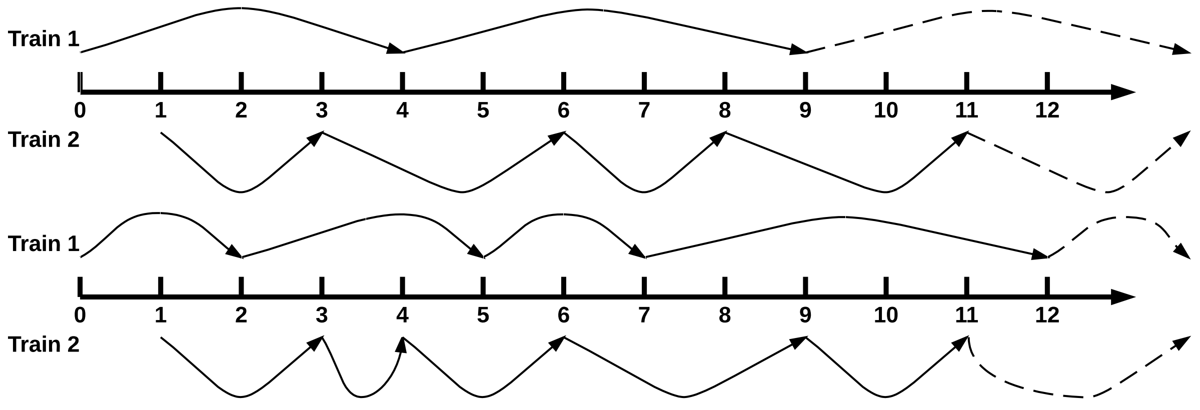

Figure 2: Two possible scenarios for the event sampling for two trains with $n = 5$

Given the randomness of the simulation approach, we repeated each experiment five time by introducing five different errors of each kind. The time (x/y) presented in Table 1 corresponds to the arithmetic mean (x) and the standard deviation (y) between their execution time.

We can make several observations from this table. Firstly, we can see that the simulation approach detects thoroughly errors in the application data, even if exhaustiveness is not guaranteed. However, it is important to mention that the application data are robust by their design. Several forms of redundancy are implicitly comprised inside them which implies that an inconsistency do not irremediably causes a safety issue. For instance, removing an UIR statement in the route setting conditions can be covered by another contiguous condition. Therefore, introducing errors causing safety issues becomes an harder task than expected.

Besides, we can also observe that our approach detects errors significantly faster than the model checking approach. Concerning the scalability of the approach, the benchmark for Braine l'Alleud shows that the execution time for a larger station do not increase that much compared to Namêche. Although experiments on larger data set have not been done yet, the possibility of a full parallelisation strengthens the scalability of the approach.

Indeed, the verification by simulation can be almost entirely paralleled without any overhead if the simulation time is not too short. The intuition behind this assumption is that the train traffic occurring today does not influence the train traffic occurring 10 days later. In other word, a simulation covering 20 days is identical to two simulation of 10 days. The only overhead occurring is due to the time used to parse the application data and to generate the simulation model which can be neglected. By Gustafson's law (Gustafson, 1988), we can thereby deduce that with n processors, the execution time should be divided by n.

Furthermore, before using simulation for the verification, we need to decide during how much time the simulation must be active in order to have a high expectation that the potentially errors will be successfully discovered. The first step is to know what is the correspondence between the simulation time and the real time or in other words, how many hours are covered for a simulation of one hour. To do so, let us assume the worst case of a busy station where there is an incoming train every minute during a whole day (1440 trains per day). By recording the number of routes set during the simulation, we can report the number of trains that have moved through the station and deduce how many days the simulation has covered. Following this procedure, we obtain that 1 hour of simulation for Braine l'Alleud covers approximatively 16 872 days ($\approx$ 46 years) of real interlocking operations. A simulation of 1000 years will thereby takes 22 hours of computation without resorting to parallelisation. However, the results of Table 1 shows that in practice error detection is done far more quickly that the time required to reach this period.

The major drawback of our approach is that there is no guarantee of exhaustiveness. Therefore, it is theoretically possible that there exist conflictual scenarios not covered by the simulation. To gain confidence about our model, we have designed covering tests aiming to measure which scenarios are tested. For an interlocking system, a scenario corresponds to a route request accepted whereas the station has a particular configuration. For instance, the Request Q(R_CC_102) of Listing 1 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 (Ammann and Offutt, 2008) is used to gain confidence into the quality of test suites, we also measure and report statistics related to the scenarios coverage. More exactly we record for each request the number of times it is generated, granted, and which routes were already set when granted.

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 2 summarizes this test coverage for the scenarios where a request is done when Route R_CXC_104 is set. 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.

Table 2: Test Coverage when Route R_CXC_104 is Set

Request	$\frac{\#Granted}{\#Done}\%$	Request	$\frac{\#Granted}{\#Done}\%$
DXC_092	0	EC_091	0
KXC_103	22.65	CC_102	0
CC_103	0	CGC_012	13.70
KC_101	24.37	CC_104	0
CC_101	24.85	KXC_101	24.13
KC_102	24.36	IC_011	14.51
EC_092	0	FC_091	0
KXC_104	21.89	KC_104	23.11
DC_091	12.62	KC_103	22.78
JXC_011	16.16	CXC_101	12.71
CXC_102	0	JC_012	14.75
JXC_012	16.33	DXC_091	0
CGC_011	13.69	CXC_103	0
KXC_102	24.15	DC_092	12.57
IC_012	14.05	FC_092	0
JC_011	15.38	-	-

This table summarises the proportion of times (in percent) that a request is granted after being issued. We can observe that some requests, like Q(R_DXC_092), are always refused when Route R_CXC_104 is set. In Figure 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 set.

For the other requests, they are all 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. It corresponds mainly to the routes beginning in the middle of the station (from Signals DC, DXC, ED, IC, CGC, JC or JXC in Figure 1). It is because on such locations, trains can have a route going either to left, or to right. Therefore, the probability to have a route going to a particular direction is reduced. Furthermore the requests having the lowest probability to be granted are Q(R_CXC_101), Q(R_DC_091) and Q(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 set. Similar results are observed for scenarios involving other routes, which shows that most of the scenarios are covered by the simulation.

Through the analysis of the results presented in this section, we thereby confirmed the validity of the simulation approach, its performance, and its benefits.

CONCLUSION

Verification of an interlocking system is a safety concern. Up to now, most of the research work considered for the verification were based on model checking. However, due to the state space explosion, this approach does not scale very well for large stations. In this paper, we presented a new approach of verification based on a discrete event simulation aiming to address the problems encountered by the model checking. We have first described the principles of this approach and how it can be used for a verification purpose. Besides, we have confirmed through experimental results the validity of this approach and shown that it does not suffer from the problems of the model checking approach. Furthermore, we presented other benefits of the method as the possibility of a full parallelisation in order to do a long-term simulation.

The present work focuses on defining the discrete event simulation. However, it does not show how the results of performing several simulations can be aggregated to obtain a confidence on the reliability of the entire system.

As a future work, we plan to plug our approach in a statistical model checker (SMC). The idea behind SMC is to perform several simulations of a given system, and then to use statistics (Monte Carlo, hypothesis testing, etc.) in order to reveal information on its global behavior (Legay et al., 2010; Younes and Simmons, 2006). Classical SMC approaches are well-suited to compute the probability that a system satisfies a given property. Here, we are more interested in guiding the simulation so that it reveals a rare failure of the system. Our knowledge of the system will be used to force the simulation to reach a failure, and to stop it if there is enough evidence that it will not reach a failure.

ACKNOWLEDGEMENT

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

REFERENCES

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

Busard S.; Cappart Q.; Limbrée C.; Pecheur C.; and Schaus P., 2015. *Verification of railway interlocking systems. arXiv preprint arXiv:150603554*.

Clarke E.M.; Klieber W.; Nováček M.; and Zuliani P., 2012. *Model checking and the state explosion problem*. In *Tools for Practical Software Verification*, Springer. 1–30.

Cribbens A., 1987. *Solid-state interlocking (SSI): an integrated electronic signalling system for mainline railways*. In *IEE Proceedings B (Electric Power Applications)*. IET, vol. 134, 148–158.

Eisner C., 1999. *Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard*. In *Correct Hardware Design and Verification Methods*, Springer. 99–109.

Gery E.; Harel D.; and Palachi E., 2002. *Rhapsody: A complete life-cycle model-based development system*. In *Integrated Formal Methods*. Springer, 1–10.

Gustafson J.L., 1988. *Reevaluating Amdahl's Law. Commun ACM*, 31, no. 5, 532–533. doi:10.1145/42411.42415. URL http://doi.acm.org/10.1145/42411.42415.

Hon Y.M. and Kollmann M., 2006. *Simulation and verification of UML-based railway interlocking designs*. In *Automatic Verification of Critical Systems*. 168–172.

Huber M. and King S., 2002. *Towards an integrated model checker for railway signalling data*. In *FME 2002: Formal MethodsGetting IT Right*, Springer. 204–223.

Legay A.; Delahaye B.; and Bensalem S., 2010. *Statistical Model Checking: An Overview*. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. 122–135. doi:10.1007/978-3-642-16612-9_11. URL http://dx.doi.org/10.1007/978-3-642-16612-9_11.

Müller K. and Vignaux T., 2003. *SimPy: Simulating Systems in Python. ONLampcom Python DevCenter*. URL http://www.onlamp.com/pub/a/python/2003/02/27/simpy.html?page=2.

Nash A. and Huerlimann D., 2004. *Railroad simulation using OpenTrack. Computers in railways IX*, 45–54.

OscaR Team, 2012. *OscaR: Scala in OR*. Available from https://bitbucket.org/oscarlib/oscar.

Schriber T.J.; Brunner D.T.; and Smith J.S., 2012. *How discrete-event simulation software works and why it matters*. In *Proceedings of the Winter Simulation Conference*. Winter Simulation Conference, 3.

Theeg G.; Anders E.; and Vlasenko S., 2009. *Railway Signalling & Interlocking: International Compendium*. Eurailpress. ISBN 9783777103945. URL https://books.google.be/books?id=Y-vrSAAACAAJ.

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

Winter K.; Johnston W.; Robinson P.; Strooper P.; and Van Den Berg L., 2006. *Tool support for checking railway interlocking designs*. In *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*. Australian Computer Society, Inc., 101–107.

Younes H.L.S. and Simmons R.G., 2006. *Statistical probabilistic model checking with a focus on time-bounded properties. Inf Comput*, 204, no. 9, 1368–1409. doi:10.1016/j.ic.2006.05.002. URL http://dx.doi.org/10.1016/j.ic.2006.05.002.