LightF3: A Lightweight Fully-Process Formal Framework for Automated Verifying Railway Interlocking Systems

Yibo Dong[*]
Xiaoyu Zhang[*]
East China Normal University
Shanghai, China

Yicong Xu
Chang Cai
Yu Chen
East China Normal University
Shanghai, China

Weikai Miao
Jianwen Li[†]
Geguang Pu[†‡]
East China Normal University
Shanghai, China

ABSTRACT

Interlocking has long played a crucial role in railway systems. Its functional correctness, particularly concerning safety, forms the foundation of the entire signaling system. To date, numerous efforts have been made to formally model and verify interlocking systems. However, two main problems persist in most prior work: (1) The formal description of the interlocking system heavily depends on reusing existing models, which often results in overgeneralization and failing to fully utilize the intrinsic characteristics of interlocking systems. (2) The verification techniques of current approaches may quickly become outdated, and there is no adaptable method to integrate state-of-the-art verification algorithms or tools.

To address the above issues, we present LightF3, a lightweight and fully-process formal framework for modeling and verifying railway interlocking systems. LightF3 provides RIS-FL, a formal language based on FQLTL (a variant of LTL) to model the system and its specifications. LightF3 transforms the RIS-FL model automatically to the aiger model, which is the mainstream input of state-of-the-art model checkers, and then invokes the most advanced checkers to complete the verification task. We evaluated LightF3 by testing five real station instances from our industrial partner, demonstrating its effectiveness as a new framework. Additionally, we analyzed the statistics of the verification results from different model-checking techniques, providing useful conclusions for both the railway interlocking and formal methods communities.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**.

KEYWORDS

Formal Methods, Interlocking Systems, Model Checking

[*]Both authors contributed equally to this research.
[†]corresponding author
[‡]Also with Shanghai Trusted Industrial Control Platform Co., Ltd.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3613874

ACM Reference Format:
Yibo Dong, Xiaoyu Zhang, Yicong Xu, Chang Cai, Yu Chen, Weikai Miao, Jianwen Li, and Geguang Pu. 2023. LightF3: A Lightweight Fully-Process Formal Framework for Automated Verifying Railway Interlocking Systems . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611643.3613874

1 INTRODUCTION

An interlocking system is a control system responsible for guiding the trains safely through the railway network in accordance with traffic regulations and disciplines. By continuously maintaining the state of devices , the interlocking system determines whether it is safe to perform certain operations, such as allowing the train to enter a specific track. Additionally, by controlling active elements, the interlocking system serves as a vital interface between trains and other railway components (as shown in Fig. 1). Therefore, the functional correctness of the interlocking system, especially safety correctness, is crucial to the entire signal system and must meet a high safety integrity level (SIL4) [21].

Despite the importance of ensuring safety, many railway companies still rely on manual testing and simulation due to a lack of efficient and cost-effective mechanisms for verifying safety properties. Though formal methods [34] have shown promise, the complex professional background and universal confidentiality of the railway industry make it difficult to apply these techniques. As a result, most research [12, 49] focuses on specific station cases with moderate scale or simple properties.

Moreover, prior works on formally verifying interlocking systems have mainly adopted fixed verifiers and attempted to reuse existing models like SMV [11] or transform into them [16], typically by feeding them into third-party IDEs [24]. This approach is difficult to extend and eliminates the possibility of reserving instance-oriented quantifiers along with past operators, which are both crucial components for complex practical properties not supported in existing models. Safety properties in interlocking systems, originating from a common discipline, distinguish themselves from those in other domains in that they are highly homogeneous both inter-station and intra-station. Manually writing repetitive and error-prone low-level properties falls far behind writing a few generic properties and instantiating them according to detailed data. Furthermore, without past operators, time-sensitive properties may be beyond expression. Regarding verification techniques, existing verifiers generally employ bounded model checking [6] as the core technique. While this approach is efficient in finding shallow bugs,

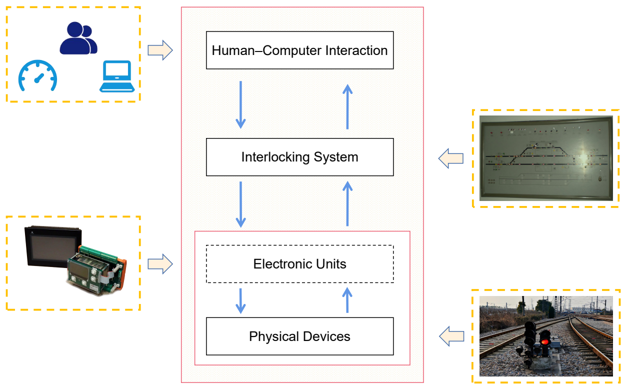


Figure 1: Electronic Based Interlocking system

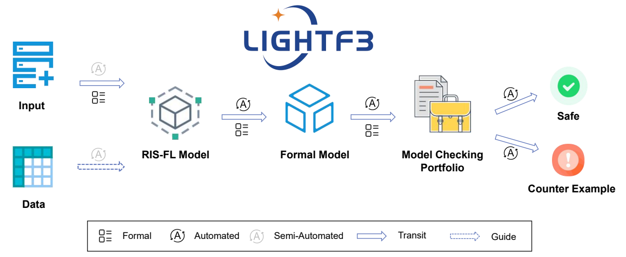


Figure 2: The Schema of LightF3

state-of-the-art verifiers like CAR [41] and IC3/PDR [10, 17, 20] have demonstrated superior performance under different circumstances. However, there is currently no convenient way to plug them in or reconfigure them according to the verification result.

In our opinion, the model of the interlocking system can be represented as an acyclic graph constructed based on the topology of the track layout, along with a set of rules that describe the relationships among the devices. The devices, each having unique IDs and various attributes, are represented as vertices, while the rules are represented as edges connecting the vertices. For example, 'A switch s_1 is in the track t_1' can be expressed as the '$BelongToTrack$' attribute of s_1 being 't_1', and the corresponding rule 's_1 should be in track t_1' will then be specified as '$s_1.BelongToTrack == t_1$'. On this basis, the interlocking system can be seen as a graph with rich information stored in its vertices. Alternatively, one can regard it as a circuit-like system with a moderate scale. The verification of such system is a prevalent subject in hardware model checking.

We propose a framework called LightF3 for automated verification of railway interlocking systems (Fig. 2). The term "Light" indicates that it does not involve translation into a more complex and general model; it also reduces the difficulty of writing formal properties, making it easy for production personnel to use. Additionally, plug-and-play is supported for any aiger-based work-of-art verifier, making the cost of trying out the latest verifier insignificant. The triple "F" denotes "Fully-process Formal Framework," which is its distinguishing feature. After writing the RIS-FL model, all subsequent procedures, including model transformation and verification, are formal. Based on the recently proposed Finite Quantifier Linear Temporal Logic(FQLTL) [15], we establish a formal language called RIS-FL (short for Railway Interlocking System Formal Language) to describe the model and write generic properties. Furthermore, we provide users with a user-friendly interface in LightF3. These generic properties do not specify particular devices and should be written in accordance with traffic regulations. As a result, they can be shared among different stations. Concrete properties are generated by instantiating them with detailed station-specific application data for further verification. With an extensible and re-configurable verification portfolio, different properties can be efficiently verified based on their aptitude.

We invited our industrial partners to try out LightF3, and with a moderate amount of effort to learn how to write formal specifications and translate their station data, they were able to successfully verify practical stations of various sizes. The largest station had around sixty tracks and fifty switches with over 200 routes. Through the use of LightF3, we helped them discover errors in their prior natural language specifications and encouraged them to clarify relevant concepts for their employees. Overall, our partnership with them was a success, and they were pleased with the results.

Novelty. We provide the following contributions:

- A fully-process formal framework LightF3 that :
  - proposes a formal language RIS-FL and allows writing formal descriptions at a moderate cost.
  - effectively transits to model checking problem.
  - can carry any latest aiger-based [5] verifiers to solve interlocking system problems.
- Investigate the performance of various model-checking techniques in interlocking contexts, conclusions of which benefit both industry and academia.
- Pose an example benchmark, which takes an interlocking system as the background, to facilitate researchers who are interested in practical interlocking problems.

2 RELATED WORK

There have been numerous efforts to apply formal methods and tools for ensuring the correctness of railway system designs [23, 34]. In the early days, general-purpose models such as UML [43], state machines (or automata) [14], and Petri-Nets [42] were used, which posed scalability issues. More successful applications emerge by introducing the B method [3], SMV [11] as the modeling language, with which powerful tools like ProB [37] and NuSMV [16] can verify the obtained models in an efficient way. Nevertheless, these methods are still too generic to depict dedicated features of interlocking systems. For example, the B method cannot handle the temporal information well while SMV lacks the (direct) support for metric temporal information and finite domains. For existing domain-specific solutions, SafeCap [29–31] aims at modeling the whole railway network, making it redundant to use on interlocking systems. Also, railML [13] is a sufficient domain language to describe an interlocking system, while it is semi-formal [46] and proposes to utilize SAML [26], which is a formal language with limited support for temporal reasoning, to formalize railML models. Finally, the ladder logic [8, 33] is widely used in commercial tools like Prover ILock [9] and SCADE [35], which may be the best option to model interlocking systems to date. However, Ladder logic may express limited temporal options by using the *latch* variables only.

Towards the specifications of interlocking systems, previous works mainly use pure propositional logic [19] or variants of LTL [39, 47] to formalize from different levels. However, most of them do not consider extending LTL to describe generic properties suitable for the same types of devices in the system, i.e., by introducing quantifiers over variables. Although [27] allows the quantifiers, they can be only affiliated to the *bound* variables for bounded model checking. Also, the LTL versions supported in the SMV language do not include quantification.

Model checking [18] and theorem proving [4] are two main verification techniques for interlocking systems. While theorem-proving B models have gained success in practice, a lot of artificial efforts are required to complete the proving. Meanwhile, model checking can be achieved automatically once the model and properties are prepared, which is more promising in the view of industrial applications. Indeed, series of works [27] rely on model checking techniques like BMC [6], K-induction [49] and IC3/PDR [17] to verify the interlocking system. However, most of them consider integrating such algorithms inside their methodology, running upon their self-defined models. The cost can become heavy when considering integrating new model-checking techniques instead of leveraging state-of-the-art third-party model checkers, e.g., SimpleCAR [38], IC3-ref [1], AVY [48] etc.

Our framework LightF3 is distinguished from others in the following aspects. (1) LightF3 uses the RIS-FL modeling language which has FQLTL underlined and is more dedicated to modeling railway systems rather than B, SMV, and SAML languages as well as the Ladder Logic; (2) Once the RIS-FL model is created, the left verification process is fully automated, including the property/constraint instantiation that is specific to interlocking systems; (3) Finally, LightF3 is lightweight and flexible because it leverages the Aiger [5] format as the input for model checking, which is the mainstream nowadays and therefore is easy to import new model-checking techniques as an aid to efficient verifying.

3 PRELIMINARIES

3.1 First-Order Logic

The syntax of first-order logic is defined relative to a signature σ, which consists of a set of constant symbols, a set of function symbols, and a set of predicate symbols. Each function and predicate symbol has an *arity* $k > 0$. Formally, a first-order logic formal ϕ has the form of

$$\phi ::= t \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x\ \phi \mid \forall x\ \phi$$

where the term t can be a variable, constant symbol, or k-ary function symbol $f(t_1, \dots, t_k)$. The symbol $\exists$ and $\forall$ refer to *existential* and *universal quantifier* separately.

Given a signature σ, a σ-structure $\mathcal{A}$ consists of:

- a non-empty set $U_{\mathcal{A}}$ called the universe of the structure;
- for each k-ary predicate symbol P in σ, a k-ary relation $P_{\mathcal{A}} \subseteq \underbrace{U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}}}_{k}$;
- for each k-ary function symbol f in σ, a k-ary relation $f_{\mathcal{A}} : \underbrace{U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}}}_{k} \to U_{\mathcal{A}}$;
- for each constant symbol c, an element $c_{\mathcal{A}}$ of $U_{\mathcal{A}}$;
- for each variable x an element $x_{\mathcal{A}}$ of $U_{\mathcal{A}}$.

Given a structure $\mathcal{A}$, variable x, and $a \in U_{\mathcal{A}}$, we define the structure $\mathcal{A}_{[x\mapsto a]}$ to be exactly the same as $\mathcal{A}$ except that $x\mathcal{A}_{[x\mapsto a]} = a$. We define the value $\mathcal{A}[\![t]\!]$ of each term t as an element of the universe $U_{\mathcal{A}}$ inductively as follows:

- For a constant symbol c we define $\mathcal{A}[\![c]\!] \overset{\text{def}}{=} c_{\mathcal{A}}$;
- For a variable x we define $\mathcal{A}[\![x]\!] \overset{\text{def}}{=} x_{\mathcal{A}}$;
- For a term $f(t_1, ..., t_k)$, where f is a k-ary function symbol and $t_1, ..., t_k$ are terms, we define $\mathcal{A}[\![f(t_1, ..., t_k)]\!] \overset{\text{def}}{=} f_{\mathcal{A}}(\mathcal{A}[\![t_1]\!], ..., \mathcal{A}[\![t_k]\!])$.

We define the satisfaction relation $\mathcal{A} \vDash \phi$ between a σ-structure $\mathcal{A}$ and σ-formula ϕ by induction over the structure of formulas.

- $\mathcal{A} \vDash P(t_1, ..., t_k)$ iff $(\mathcal{A}[\![t_1]\!], \dots, \mathcal{A}[\![t_k]\!]) \in P_{\mathcal{A}}$;
- $\mathcal{A} \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{A} \vDash \phi_1$ and $\mathcal{A} \vDash \phi_2$;
- $\mathcal{A} \vDash \phi_1 \vee \phi_2$ iff $\mathcal{A} \vDash \phi_1$ or $\mathcal{A} \vDash \phi_2$;
- $\mathcal{A} \vDash \neg\phi_1$ iff $\mathcal{A} \nvDash \phi_1$;
- $\mathcal{A} \vDash \exists x\ \phi_1$ iff there exists $a \in U_{\mathcal{A}}$ such that $\mathcal{A}_{[x\mapsto a]} \vDash \phi_1$;
- $\mathcal{A} \vDash \forall x\ \phi_1$ iff $\mathcal{A}_{[x\mapsto a]} \vDash \phi_1$ for all $a \in U_{\mathcal{A}}$;

3.2 Linear Temporal Logic

LTL was introduced into computer science in the 1970s and is used in various fields [24, 25, 28, 45, 50]. It uses temporal operators to express the behavioral constraints that need to be satisfied by a system at each moment in the past, present, and future. Let AP be a set of atomic properties, we can define the syntax of LTL formulas:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X\ \phi_1 \mid \phi_1\ U\ \phi_2 \mid \phi_1\ R\ \phi_2$$

where $p \in$ AP is an atomic proposition; ϕ is an LTL formula; $\top, \bot$ denote *true* and *false*, and X , U , R are temporal operators, representing 'Next', 'Until' and 'Release' respectively.

In LTL, U and R are dual operators, which means $\phi_1 U \phi_2 \equiv \neg(\neg\phi_1 R \neg\phi_2)$. Also, the following abbreviations are widely used in LTL: $F\ a \equiv \top\ U\ a$ and $G\ a \equiv \bot\ R\ a$.

Let $\Sigma = 2^{AP}$ be the set of alphabet and a trace $\xi = \omega_0\omega_1\omega_2...$ be an infinite sequence in Σ^ω. For ξ and $k \geq 0$ we use the following denotations:

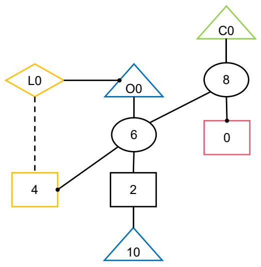
- $\xi[k]$: the k th element of ξ
- $\xi^k = \omega_0\omega_1...\omega_{k-1}$, the prefix of the trace
- $\xi_k = \omega_k\omega_{k+1}...$, the later part of the trace

Therefore, $\xi = \xi^k \xi_k$. The semantics of LTL formulas with respect to the infinite trace ξ is then given by:

- $\xi \models \top$ and $\xi \not\models \bot$;
- $\xi \models p$ iff $p \in \xi[0]$ where p is an atom;
- $\xi \models \neg\phi$ iff $\xi \not\models \phi$;
- $\xi \models \phi_1 \wedge \phi_2$ iff $\xi \models \phi_1$ and $\xi \models \phi_2$;
- $\xi \models X\ \phi$ iff $\xi_1 \vDash \phi$;
- $\xi \models \phi U \psi$ iff $\exists i \geq 0, \xi_i \models \psi$, and $\forall 0 \leq j < i, \xi_j \models \phi$;
- $\xi \models \phi R \psi$ iff either $\forall i \geq 0, \xi_1 \models \psi$ or, $\exists i \geq 0$, $\xi_i \models \phi \wedge \psi$ and $\forall 0 \leq j \leq i,\ \xi_j \models \psi$.

```
1   aag 4 1 1 1 2 0 1
2   2
3   4 7
4   6
5   8
6   6 2 5
7   8 6 1
```

(a) ASCII AIGER



(b) Visualization

Figure 3: Example of an AIG

Above is the standard LTL. Now we define LTL$_P$ as LTL with past operators, e.g., PRE and Since, so that we can make statements on past time instants.

3.3 Model Checking

3.3.1 And-Inverter Graph. An And-Inverter Graph (AIG) is a directed, acyclic graph designed to represent gate-level hardware circuits [5]. It is a compact and simple sequential hardware model designed for model checking competition, as there are only three basic components inside an AIG: AND gates, inverters and latches.

Fig. 3(a) and 3(b) show the ASCII AIGER format and the represented circuit of an AIG. Figure 3(a)'s first line is the header, denoted by 'aag' for ASCII format, followed by counts for different components: total components, inputs, latches, outputs, AND gates, bad states, and invariant constraints, in that order. Components are identified by positive even numbers; with the subsequent odd number representing components with inverter gates. Especially, the constants 0 and 1 are preserved to represent $\top$ and $\bot$.

Beginning from Line 2, each of the inputs, latches, outputs, constraints, bad properties, and and-gates are listed in order. For example, the second line says that the input is denoted as the literal 2, while the third line shows that 4 is the latch and 7 is preserved as the value of the latch in the next cycle. More information are referred to [5].

Besides circuits, AIG can also be used to formulate SAT and model-checking problems. Most modern model checkers support AIG as their input, and many other forms (like SMV [16]) can be easily translated to AIG through tools in AIG release.

3.3.2 Model Checking Techniques. Given a transition system $Sys = (V, I, T)$ (the model) and a safety property P, model checking answers the question that whether all behaviors of the transition system satisfy the property. If not, a trace from the initial state to the bad state, in which the property is violated, will be returned as a counterexample. Otherwise, an invariant containing the initial state can be found, indicating that the model satisfies the property.

State-of-the-art model-checking techniques like BMC [7], IMC [44], IC3/PDR [10, 17, 20] and CAR [41], are all SAT-based and there isn't a single technique that can dominate others. BMC is the first technique to introduce SAT [40] into model checking and is quite efficient in bug-finding, but it is an incomplete approach as it can't prove the property. IMC complements BMC by computing interpolants and maintaining an over-approximate state sequence inside BMC, which enables the construction of a correctness proof. Compared to BMC and IMC, IC3/PDR and CAR only unroll the transition relation at most once, which reduces the difficulty of a single SAT query but increases the total amount of SAT queries. Notably, CAR has two versions, i.e., Forward CAR and Backward CAR, which distinguish from each other by search strategy for the verification. Often, Forward CAR is better to prove correctness while Backward CAR is more advantageous in finding bugs [38, 41].

3.4 Verification of Interlocking System

In railway signaling, interlocking refers to the arrangement of signal apparatus to prevent conflicting movements, such as arranging signals and signal appliances properly. The properties in the interlocking system can be divided into two categories: safety properties and liveness properties [2]. Safety properties aim to ensure that no unsafe conditions occur, while liveness properties focus on ensuring that the train eventually leaves the station. The primary goal of interlocking is to ensure safety. Basic safety goals are usually specified at a high abstract level, and various approaches can be used to implement them at a specific station. For a specific station, the basic safety goals are concretized at multiple levels. For instance, abstract safety rules are first categorized by the type of devices and then instantiated to concrete properties of specific devices according to the configuration data. In industrial practice, a control table [22] is created to represent the possible operations of various components in the railway yard and enforce the principles and constraints.

An example to define basic safety goals from Denmark [36] is:

- Trains/shunt movements must not collide.
- Trains/shunt movements must not derail.
- Trains/shunt movements must not collide with authorized vehicles or human beings crossing the railway.
- Protect railway employees from trains.

As to formal verification, we take only the prior three goals into consideration.

4 LIGHTF3 FRAMEWORK

In this section, we present the structure of LightF3. Firstly, we discuss the general workflow, followed by an illustrative example, and then introduce each component separately. The general organization is depicted in Fig. 4.

The input of the system can be divided into five parts, as shown in the figure. These parts are then translated to generate a RIS-FL model, along with a station-specific domain interpretation. The interpretation guides the instantiation of properties and constraints, eliminating the quantifiers and creating concrete properties about specific devices. Typically, one generic property corresponds to several devices of the same type. These concrete properties together with the RIS-FL model are then transformed into a common AIGER model and passed to the model-checking portfolio. If the property does not hold for the target device, a counterexample is generated.

4.1 Illustrating Example

We would pose an illustrating example of an interlocking system here. The raw materials obtained in this study cannot be shared subject to confidential agreements. Therefore, we would give a preprocessed model and omit the detailed generation process of

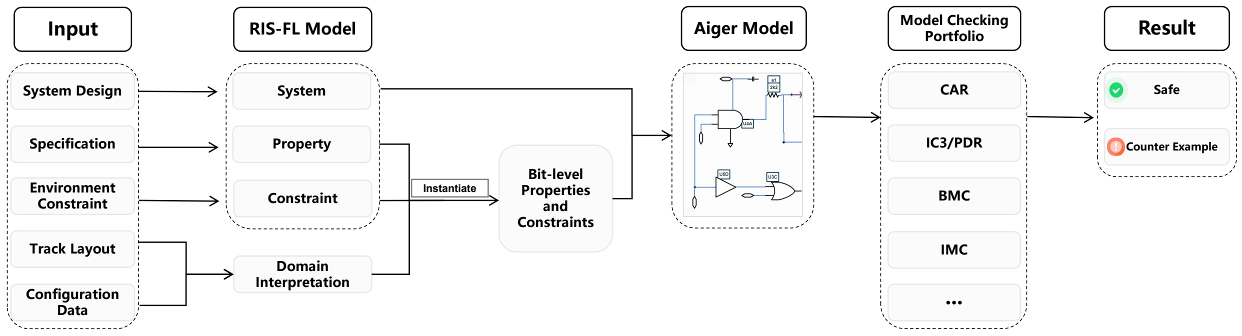


Figure 4: LightF3 Framework Work-Flow

domain interpretation. With an example track layout (Fig. 5), we try to verify a simple property:

Example 4.1 (Natural Language Description). If a track that contains switches is released, the following properties should hold:

- The track is logically clear for at least 3 seconds.
- The track is in route released state.
- The track is not route locked or occupied.

And the relevant model may look like this:

Example 4.2 (Relevant Input Model).

- P1-R = (P1-LCE & ¬P1-RR & P1-RLO)
- P3-R = (P3-LCE & ¬P3-RR & P3-RLO)
- P1-LCE = GLOBAL [0,3](P1-A)
- P3-LCE = GLOBAL [0,3](P3-A)
- P1-RR = (P1-B & P1-C)
- P3-RR = (P3-B & P3-C)
- P1-RLO = (¬P1-D & P1-E || P1-F & P1-RLO & ¬P1-LCE)
- P3-RLO = (¬P3-D & P3-E || P3-F & P3-RLO & ¬P3-LCE)
- ...

This is a generic property that all tracks should follow, therefore the outer wrapper of the property would be like "ALL track (…)", which is a syntax sugar. Besides, taking future debugging into consideration, it is suggested to split the property into sub-properties to avoid unrevealed failure during calculation because of the short circuit characteristics (yet this is not a must). Therefore, the formal properties would be:

Example 4.3 (Generic Properties in RIS-FL).

- SubRequirement-1 := ALL track (
  SOME switch (BelongToTrack(switch,track)) &
  Released(track) ⟶ LogicallyClearElapsed(track));
- SubRequirement-2 := ALL track (
  SOME switch (BelongToTrack(switch,track)) &
  Released(track) ⟶ RouteReleased(track));
- SubRequirement-3 := ALL track (
  SOME switch (BelongToTrack(switch,track)) &
  Released(track) ⟶ ¬ RouteLockedOccupied(track));

Then the formula would be instantiated according to the domain interpretation. As shown in topology (Fig. 5), the whole set of tracks in this system is {T1, T2, T3, T4}. We can easily conclude from the

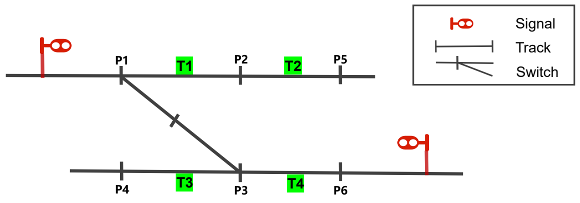


Figure 5: An Example of Station Layout

Table 1: belongToTrack

switch	track
...	...
P1	T1
P3	T3
...	...

Table 2: State function mapping

Function name	Literal name
Released(track)	${track}-(R)
LogicallyClearElapsed(track)	${track}-(LCE)
RouteReleased(track)	${track}-(RR)
RouteLockedOccupied(track)	${track}-(RLO)

concrete properties that it always holds for a track that has no switches belonging to it. According to the truth table (Table. 1), only T1, T3 needs further consideration. "Released()" and "LogicallyClearElapsed()" are both state functions and should be checked chronologically. Therefore, we would transform them using the given mapping rule (Table. 2).

The result afterward shall be as follows:

Example 4.4 (Concrete Properties).

- SubRequirement-1-T1 := (¬ (P1-R) || P1-LCE)
- SubRequirement-1-T3 := (¬ (P3-R) || P3-LCE)
- SubRequirement-2-T1 := (¬ (P1-R) || P1-RR)
- SubRequirement-2-T3 := (¬ (P3-R) || P3-RR)
- SubRequirement-3-T1 := (¬ (P1-R) || P1-RLO)
- SubRequirement-3-T3 := (¬ (P3-R) || P3-RLO)

Afterward, we would combine the concrete properties and the input model together to generate an aiger model. To achieve this, We translate all the 'or' and 'imply' statements into 'and' statements together with 'not'. For example, we would translate "$\neg a||b$" into "$\neg(a\&\neg b)$". We also introduce intermediate variables to represent the temporary variables, and latches are introduced to store values for timed expressions. For example, "*PRE a*" implies that a should be stored in a latch, and therefore can be later referred to. The detailed translation will be shown in Section 4.4.

Once the aiger model is generated, we use a verification portfolio to verify the model. Here we use IC3-ref [1] as the verifier, which quickly determines that all properties are safe.

4.2 Framework Inputs

The inputs of a formal verification system can be generally divided into two parts: the model of the system and the corresponding specifications. In LightF3, however, due to the distinguishing characteristics of interlocking systems, it has additional data part and environment constraints part.

The system model describes the correlations between devices and how the attribute of one device changes in the next cycle based on all devices, which can be represented using LTL_P formulas. Meanwhile, the properties and constraints are expressed in FQLTL. Although they may appear similar, properties describe the whole interlocking system, while constraints serve as preconditions to compress the state space and speed up verification. The track layout and configuration data are maintained by the station and form the later domain interpretation after a transformation process.

4.3 RIS-FL Model

LTL is widely used to express the behavioral constraints satisfied by a system at each moment. However, in interlocking specifications, the domain of entities referred to by the properties is limited. For example, all the devices whose type is "route" have to satisfy a particular property P_1, LTL can only determine whether P_1 is satisfied, but not in the restricted domain "route R". One step further, practical properties need to specify in a finite-time manner. We also need to add concrete time-range qualifiers to the binary temporal operators for practical use.

FQLTL is a recently proposed logic [15], which can express relational and temporal properties so as to restrict the finite domain of devices described in LTL specification. FQLTL is accessible to describe a system with multiple, interrelated devices, the syntax and semantics of which are defined as follows.

Definition 4.5 (Syntax of FQLTL *formulas).* A legal FQLTL formula ϕ has the following syntax:

$$\begin{aligned}\phi ::= t \mid (\phi) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \\ P(t_1, ..., t_k) \mid \mathsf{ALL}\ x \cdot \phi \mid \mathsf{SOME}\ x \cdot \phi \mid \\ \mathsf{PRE}\ \phi \mid X\phi\ \mid \phi\ U_{[m_1,m_2]}\ \phi \mid \phi\ S_{[m_1,m_2]}\ \phi;\end{aligned}$$

In the above, $t, t_1, ..., t_k$ are the *terms* and P is the predicate symbol as defined in First-Order Logic. ALL is the universal quantifier, which is a syntax sugar of $\forall$, while SOME is the existential quantifier, which is a syntax sugar of $\exists$. Meanwhile, PRE, X, U, and S are all temporal operators, where PRE is the Previous period operator and X means the neXt period; U is the Until operator, and S is the Since (past) operator. And the underlined time range is the qualifier mentioned above. In particular, we use $\phi_1\ R\ \phi_2$ to denote $\neg(\neg\phi_1 U \neg\phi_2)$, i.e., where R is the dual operator of U; and we use the usual abbreviations:

- $G_{[m_1,m_2]}\ \phi = \bot\ R_{[m_1,m_2]}\ \phi$
- $F_{[m_1,m_2]}\ \phi = \top\ U_{[m_1,m_2]}\ \phi$

Definition 4.6 (Semantics of FQLTL *formulas).* Let ξ be an infinite trace, σ be a signature and $\mathcal{A}$ be the corresponding σ-structure such that the universe of $\mathcal{A}$, i.e., $U_{\mathcal{A}}$, is a finite set. Then the semantics of FQLTL formulas are interpreted over the tuple $\langle\xi, \mathcal{A}, i\rangle$ such that:

- $\langle\xi, \mathcal{A}, i\rangle \vDash t$ iff $\mathcal{A}[\![t]\!] = true$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash P(t_1, ..., t_k)$ iff $(\mathcal{A}[\![t_1]\!], \dots, \mathcal{A}[\![t_k]\!]) \in P_{\mathcal{A}}$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash (\phi)$ iff $\langle\xi, \mathcal{A}, i\rangle \vDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \neg\phi$ iff $\langle\xi, \mathcal{A}, i\rangle \nvDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \phi \wedge \psi$ iff $\langle\xi, \mathcal{A}, i\rangle \vDash \phi$ and $\langle\xi, \mathcal{A}, i\rangle \vDash \psi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \phi \vee \psi$ iff $\langle\xi, \mathcal{A}, i\rangle \vDash \phi$ or $\langle\xi, \mathcal{A}, i\rangle \vDash \psi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \phi \rightarrow \psi$ iff $\langle\xi, \mathcal{A}, i\rangle \nvDash \phi$ or $\langle\xi, \mathcal{A}, i\rangle \vDash \psi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \mathsf{ALL}\ x \cdot \phi$ iff $\forall\ a \in U_{\mathcal{A}}, \langle\xi, \mathcal{A}_{[x\mapsto a]}, i\rangle \vDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \mathsf{SOME}\ x \cdot \phi$ iff $\exists a \in U_{\mathcal{A}}, \langle\xi, \mathcal{A}_{[x\mapsto a]}, i\rangle \vDash \phi$;
- $\mathcal{A} \vDash (t_1 = t_2)$ iff $\mathcal{A}[\![t_1]\!] = \mathcal{A}[\![t_2]\!]$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \mathsf{PRE}\ \phi$ iff $i > 0$ *and* $\langle\xi, \mathcal{A}, i-1\rangle \vDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash X\ \phi$ iff $i \geq 0$ *and*$\langle\xi, \mathcal{A}, i+1\rangle \vDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \phi\ U_{[m_1,m_2]}\ \psi$ iff $m_1 \leq i \leq m_2$ and there is $i \leq j \leq m_2$ s.t. $\langle\xi, \mathcal{A}, j\rangle \vDash \psi$ and for all $i \leq k < j, \langle\xi, \mathcal{A}, k\rangle \vDash \phi$;
- $\langle\xi, \mathcal{A}, i\rangle \vDash \phi\ S_{[m_1,m_2]}\psi$ iff $m_1 \leq i \leq m_2$ and there is $m_1 \leq j \leq i$ s.t. $\langle\xi, \mathcal{A}, j\rangle \vDash \psi$ and for all $j < k \leq i, \langle\xi, \mathcal{A}, k\rangle \vDash \phi$.

The tuple $\langle\xi, \mathcal{A}, i\rangle \models \phi$ means ϕ holds in $\langle\xi, \mathcal{A}\rangle$ at step i. In particular, we define $\langle\xi, \mathcal{A}\rangle \models \phi$ iff $\langle\xi, \mathcal{A}, 0\rangle \models \phi$.

Since the universe $U_{\mathcal{A}}$ of $\mathcal{A}$ is restricted to be finite, the motivation comes up straightforwardly that the quantifiers can be eliminated for further processing. We call FQLTL without quantifiers LTL_P. It differs from LTL in that it supports past-time temporal operators. The syntax and semantics of LTL_P are similar to those of FQLTL, we just omit them here.

*Definition 4.7 (*FQLTL *Instantiation).* For an FQLTL formula ϕ with the signature σ. Let $\mathcal{A}$ be its σ-structure and $U_{\mathcal{A}}$ be the universe in $\mathcal{A}$. The instantiation of ϕ under $\mathcal{A}$, denoted as $I(\phi)$, is an LTL formula such that

- $I(P(t_1, \dots, t_k)) = \top$ iff $(\mathcal{A}[\![t_1]\!], \dots, \mathcal{A}[\![t_k]\!]) \in P_{\mathcal{A}}$; Otherwise, $I(P(t_1, \dots, t_k)) = \bot$;
- $I(\phi_1 \wedge \phi_2) = I(\phi_1) \wedge I(\phi_2)$;
- $I(\phi_1 \vee \phi_2) = I(\phi_1) \vee I(\phi_2)$;
- $I(\phi_1 \rightarrow \phi_2) = \neg I(\phi_1) \vee I(\phi_2)$;
- $I(\neg\phi) = \neg I(\phi)$;
- $I(\mathsf{PRE}\ \phi) = \mathsf{PRE}\ I(\phi)$;
- $I(X\phi) = X\ I(\phi)$;
- $I(\phi_1\ S_{[m_1,m_2]}\ \phi_2) = I(\phi_1)\ S_{[m_1,m_2]}\ I(\phi_2)$;
- $I(\phi_1\ U_{[m_1,m_2]}\ \phi_2) = I(\phi_1)\ U_{[m_1,m_2]}\ I(\phi_2)$;
- $I(\mathsf{ALL}\ x \cdot \phi) = \bigwedge_{a\in U_{\mathcal{A}}} I(\phi_{[x\mapsto a]})$, where $\phi_{[x\mapsto a]}$ is obtained from ϕ by replacing x to a;
- $I(\mathsf{SOME}\ x \cdot \phi) = \bigvee_{a\in U_{\mathcal{A}}} I(\phi_{[x\mapsto a]})$, where $\phi_{[x\mapsto a]}$ is obtained the same as above.

Based on FQLTL, the formal language for railway interlocking systems can be defined as follows:

Definition 4.8 (RIS-FL). An interlocking system is a tuple ($Model$, $Props$, $Constraints$) such that $Props$ and $Constraints$ are FQLTL formulas, and $Model$ is ($\{Device\}, Topology, \{Rule\}$) where

- $Device := (Type, ID, \{Attr\})$
- $Topology$:= (Vertex = $\{Device\}$, Edge := $\{(Device, Device)\}$)
- $Rule$ is an LTL$_P$-formula over $2^{\{\{Type\}\times\{ID\}\times\{Attr\}\}}$.

To illustrate, we give a simple example here.

Example 4.9 (Describe properties using LTL$_P$ *formulas).* Let Devices be $\{Track1, Track2\}$, and the universal domain be:

- {Type} := {Track}
- {ID} := {1,2}
- {Attr} := {Released, Locked}

then a trivial informal property:
"$If\ a\ track\ is\ released,\ then\ it\ is\ not\ locked.$"
is corresponding to the LTL$_P$ Rules:

- Track1Released → ¬ Track1Locked
- Track2Released → ¬ Track2Locked

4.4 Model Transformation

And-Inverter Graphs (AIGs) is a compact form to formulate model checking problems [5]. To construct an AIGER model, it is necessary to identify the system's temporal constraints and subsequently map them into components in AIGER. This process involves two main steps. The first step instantiates generic properties into concrete ones, eliminating all the quantifiers present in the properties.

The property instantiation procedure primarily relies on a recursive calculation, with assistance from the domain interpretation. It takes as input the expression, keeps a present mapping table from type to particular device, and generates a (S, TE) tuple, where S is the status of calculation represented in a three-valued Bool (True, False or Undetermined) and TE is the timed Boolean expression. The rough idea is expressed in pattern-matching-style Pseudo code in Algorithm 1, where we focus on the Boolean expression here for simplicity, leaving the calculation of status implicit.

Depending on the expression type, the instantiation procedure does correlative calculation. As to the logic expression, it just calculates the status using three-valued Boolean logic. If the status is either True or False, then the result expression TE should be empty. Otherwise, TE will be concatenated together. Temporal operators are reserved for later transformation while recursively calculating sub-expressions. About quantifiers, it uses domain interpretation to enumerate device instances and concatenates them together likewise. Regarding function calls, operations are performed according to detailed function types: either mapping to a literal with the given rule, or returning a Boolean value representing whether the relationship holds.

After instantiation, if the status S is true, that means this property is bound to satisfy regardless of time and does not need further checking. We would leave it out. Otherwise, if the status S is False, that means it would never be satisfiable, though this should hardly happen in real cases, we would just put a placeholder "*ReservedLiteral* & ¬ *ReservedLiteral*" in it for robustness. Meanwhile, if the status is Undetermined, the corresponding concrete properties are then generated.

Algorithm 1: Property Instantiation Algorithm

```
Input: A FQLTL property $p$, Main Device Type $t$, Device
       Domain $D$
Output: A set of corresponding LTL$_P$ properties $L$
$L := \{\}$
foreach $dev_i$ in $D$ do
    $item \leftarrow \{t := dev_i\}$
    $l_i \leftarrow INST(p, item)$
    $L \leftarrow L \cup l_i$
function INST($p, item$)
match $p$ with
| $All\ typ\ p_1 \Rightarrow$
    $\{dev\} \leftarrow getdevice(typ, D)$
    $return\ \bigwedge(INST(p_1, item\ \cup\ typ := dev_i))$
| $Some\ typ\ p_1 \Rightarrow$
    $\{dev\} \leftarrow getdevice(typ, D)$
    $return\ \bigvee(INST(p_1, item\ \cup\ typ := dev_i))$
| $p_1\ \wedge\ p_2 \Rightarrow$
    $return\ INST(p_1, item)\ \wedge\ INST(p_2, item)$
| $p_1\ \vee\ p_2 \Rightarrow$
    $return\ INST(p_1, item)\ \vee\ INST(p_2, item)$
| $\neg p_1\ \Rightarrow\ \ return\ \neg\ INST(p_1, item)$
| $PRE\ p \Rightarrow return\ PRE\ Inst(p)$
| $X\ p \Rightarrow return\ X\ Inst(p)$
| $p_1\ U_{[t_1,t_2]}\ p_2 \Rightarrow$
    $return\ INST(p_1)\ U_{[t_1,t_2]}\ INST(p_2)$
| $p_1\ S_{[t_1,t_2]}\ p_2 \Rightarrow$
    $return\ INST(p_1)\ S_{[t_1,t_2]}\ INST(p_2)$
| $default \Rightarrow$
    $(*\ use\ domain\ interpretation\ to\ calculate\ *)$
```

Algorithm 2: Generate Aiger

```
Input: $Model\ m,\ Property\ set\ P,\ Constraint\ set\ C$
Output: $Aiger\ Model\ A$
A $\leftarrow$ {}
foreach $r_i$ in $m.rules$ do
    $A \leftarrow A\ \cup\ trans(r_i)$
foreach $p_i$ in $P$ do
    $A \leftarrow A\ \cup\ trans(INST(p_i))$
    $Mark(p_i, `Property', A)$
foreach $c_i$ in $C$ do
    $A \leftarrow A\ \cup\ trans(INST(c_i))$
    $Mark(c_i, `Constraint', A)$
```

To translate propositional logic to a combination of basic components is trivial (Algorithm 2, 3). The basic idea is to use appropriate latches whenever a time operator is used. For example, a 'PRE' operator means to fetch the prior cycle's value and can be represented as a latch in aiger. Similar operations can be done with this approach. Combining the model expressions and concrete properties, the aiger model is finally produced. Notably, in Algorithm 3, function 'Var' maps an LTL$_P$ formula to an integer; 'And (c, a, b)' creates an and-gate whose output is c and inputs are a, b; 'Latch (a, b)' creates a latch whose current and next values are a, b respectively.

4.5 Verification Portfolio

Over the past few decades, model checking techniques have undergone significant evolution, enabling the solution of increasingly complex problems. Although BMC has been the go-to technique for hardware model checking, it has certain drawbacks, such as the inability to prove the absence of errors without special handling [7]. Recently, new techniques such as CAR, IMC, IC3/PDR have emerged, and while they may produce a larger number of SAT queries than BMC, most of them can be handled well by modern SAT solvers. Each of these techniques has its own strengths and weaknesses, and there is no clear winner. For example, IC3 can solve instances that BMC cannot and vice-versa [17]; The same is true among AVY [48], QUIP [32] and IC3. BMC outperforms IMC on unsafe instances [6]; CAR is able to solve some instances that are not solved by any other techniques, but not all [41].

Therefore, a verifier portfolio may be preferable to relying on a specific verifier. However, few prior works have employed the latest model-checking techniques for interlocking systems. In our framework, LightF3, we can easily add any latest aiger-based verifier without additional cost. This allows us to effectively solve practical interlocking problems using the latest model-checking techniques and provide rich feedback for future design.

Algorithm 3: The Trans Procedure

```
Input: $LTL_P\ formula\ \phi$
Output: $Aiger\ Model\ A$
$A \leftarrow$ match $\phi$ with
| atom $\Rightarrow\ \emptyset$
| $\neg\phi_1 \Rightarrow$
  $And(Var(\neg\phi_1), \neg Var(\phi_1), True)\ \cup\ trans(\phi_1)$
| $\phi_1 \vee \phi_2 \Rightarrow And(Var(\neg(\phi_1 \vee \phi_2)), Var(\neg\phi_1), Var(\neg\phi_2))$
  $\cup\ trans(\phi_1) \cup\ trans(\phi_2)$
| $\phi_1 \wedge \phi_2\ \Rightarrow$
  $And(Var(\phi_1 \wedge \phi_2), Var(\phi_1), Var(\phi_2))$
  $\cup\ trans(\phi_1) \cup\ trans(\phi_2)$
| $\phi_1 \rightarrow \phi_2 \Rightarrow$
  $trans(\neg\phi_1 \vee \phi_2)$
| $\phi_1 \leftrightarrow \phi_2\ \Rightarrow$
  $trans((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$
| $PRE\ \ \phi_1\ \Rightarrow$
  $Latch(Var(PRE\ \phi_1), Var(\phi_1)) \cup\ trans(\phi_1)$
| $X\ \ \phi_1\ \Rightarrow$
  $Latch(Var(\phi_1), Var(X\ \phi_1)) \cup\ trans(\phi_1)$
| $\phi_1\ U_{[t_1,t_2]}\ \phi_2 \Rightarrow$
    match $t_2$ with
    | $t_1 \Rightarrow trans(\phi_2)$
    | _ $\Rightarrow trans(\ X^{t_1}\ (\phi_2 \vee (\phi_1 \wedge X(\phi_1\ U_{[t_1+1,t_2]}\ \phi_2))))$
| $\phi_1\ S_{[t_1,t_2]}\ \phi_2 \Rightarrow$
    match $t_1$ with
    | $t_2 \Rightarrow trans(\phi_2)$
    | _ $\Rightarrow trans(PRE^{t_2}(\phi_2 \vee (\phi_1 \wedge PRE(\phi_1 S_{[t_1,t_2-1]}\phi_2))))$
```

5 EVALUATION

With a friendly interface, the bar of writing formal properties is lowered to a great extent. However, this may possibly incur an increase in transformation costs. Besides, the extensible design allows any state-of-the-art verifiers to give it a try on interlocking problems. For our evaluation, we are interested in testing whether the model transformation is an efficiency bottleneck that causes failure, and how different verifiers behave in interlocking contexts.

Table 3: Station Size

Station	Track	Route	Switch	Signal
Alice	14	8	3	12
Bob	56	70	14	37
Charlie	171	503	87	152
David	151	720	79	145
Eve	151	499	74	140

5.1 Evaluation Setup

We run experiment on a cluster of servers, which is equipped with an Intel Xeon Gold 6132 14-core processor at 2.6GHz and 96GB RAM. And the version of the operating system is Red Hat 4.8.5-16.

We conducted the experiment to evaluate the performance of LightF3 using five sample stations of varying sizes (Table. 3). For testing purposes, we utilized a set of 206 abstract properties. **To ensure confidentiality, we renamed the stations based on the requirements of our industrial partner.** Except for Station Eve which has 262 unsafe concrete properties among 8002 ones, all the other concrete properties are safe. This corresponds to the industrial fact that most properties to be verified do hold. The experiment is conducted serially, though some verifiers support parallel like IC3.

5.2 Evaluation Results

RQ1: What is the cost for model transformation in LightF3?

We set three checkpoints in the life cycle, by which we record the time consumption during each period. Property instantiation and aiger generation together make up the model transformation procedure, as is shown in Fig. 6.

Trivially, the complexity of instantiation is utmost $O(\prod_{i=1}^{Q} n_i)$, where Q is the count of quantifiers and n_i is the sum of devices of target type. With caching strategy and limited max parameter count, the complexity can be lower to $O(\sum_{j=1}^{n_{rF}} (\prod_{i=1}^{p_j} n_{i,j}))$, where n_{rF} means the number of relative functions, p_j the parameter of the j-th function and $n_{i,j}$ means the number of the i-th parameter in the j-th function. The short circuit characteristics in logic computation further reduce the amount of calculation. As to aiger generation, the complexity is linear with respect to amount of literals, which grows at an approximate linear velocity empirically. In summary, the model transformation procedure requires a moderate amount of time and exhibits slow growth as the model scales up.

In verification, however, model checking techniques require traversing all possible routes and states which is at least polynomial in complexity, and even degrade into exponential as the problem gets more complex. Though some implementations like Backward CAR do behave well, the time cost in most implementations grows at a high rate with respect to amount of literals.

From the result shown in Fig. 6, we can see that time consumption of model transformation grows with scale at a low speed, while

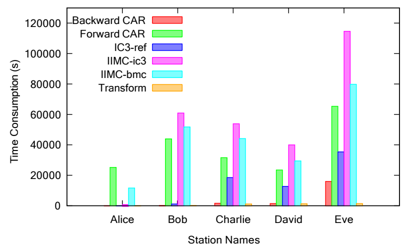


Figure 6: Sum of Time Consumption

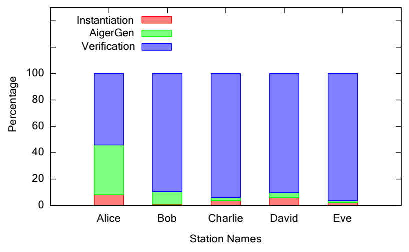


Figure 7: Time Consumption Proportion with IC3-ref

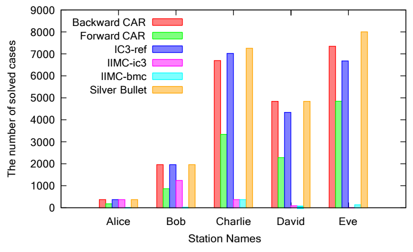


Figure 8: Sum of Solved Cases

verification time grows rapidly. Besides, we take IC3-ref as an example, see Fig. 7. The proportion of instantiation and aiger generation are both small as to large scale stations. All these corresponds with our perception. We can conclude that either the problem is trivial to verify, under such circumstance there's no threat to fail; or the transformation takes a small percentage of time. We summarize that model transformation is not likely to become an efficiency bottleneck, let alone directly causing solving failure.

RQ2: How do different verifiers behave in the context of interlocking?

Generally speaking, Backward CAR and IC3-ref perform best in interlocking verification. They solve the most cases (Fig. 8) and consumed moderate time (Fig. 6). Forward CAR comes after the prior two, while the others can only solve a few cases as to large scale problems, and are somehow not so suitable for practical interlocking verification.

For correctness proof, BMC has an inherent drawback, not surprising IIMC-bmc not doing well. Forward CAR is designed to be better in verifying safety [38], while the result is on the contrary. We then further investigate in abstract property level within station Eve. In fact, Forward CAR does better on particular properties (Fig. 10(a)). But it is too slow or even fails on some ones, and therefore slows down the verification of whole property. Adding a supplement, Forward CAR fails to find all the counterexamples, directly leading to the failure. That is possibly the reason why Backward CAR does better than Forward CAR. We also contact maintainers for help, but so far no convincing probable cause has been found.

Backward CAR performs better than IC3-ref in most models: not only in the result obtained but also in speed. However, there do exist some exceptions, like Station Charlie, where IC3-ref can solve more cases than Backward CAR. More specifically, concretize to abstract property level, it can be seen in Fig. 10(b) that they each has its own advantage, while Backward CAR performs better on the whole. The result is in line with our expectations that no model-checking techniques can outperform others in all aspects, proving the effectiveness of the extensible design in LightF3.

Furthermore, we select all unsafe instances to shed light on bug finding. IIMC-ic3 and Forward CAR has far lower efficiency. Among the other three implementations left, BMC has an inherent talent to find counterexamples, while CAR and IC3 each can find bugs that BMC cannot. We compare them pairwise and draw the scatter diagrams (Fig. 9). As is shown, Backward CAR can find most counterexamples faster than IIMC-bmc (Fig. 9(a)), while IIMC-bmc does better on most cases than IC3-ref (Fig. 9(b)), which cannot even finish all in time. Backward CAR outperforms IC3-ref in all the counter cases (Fig. 9(c)). To summarize, Backward CAR performs best in most unsafe instances, but not all.

RQ3: How effective is it for verifiers to complement each other?

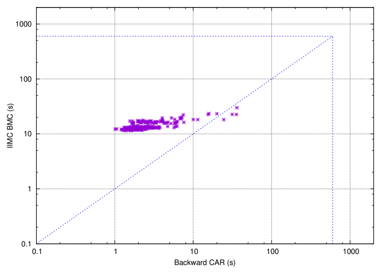
As is illustrated in RQ2, Backward CAR and IC3-ref have their advantages in overall performance, while IIMC-bmc is sometimes the best in finding counter examples. We try to use IC3-ref and IIMC-bmc to complement Backward CAR. We first union all the cases solved by each verifier and get the ceiling, with which we try omitting and backtracking to eliminate those of few contribution. Once there is nothing to omit, we get the minimal useful portfolio.

In Fig. 11, with the same solving order and time limit, IC3-ref and IIMC-bmc moderately enhance the problem-solving capability, albeit to a limited extent. Besides, adding other verifiers to the portfolio cannot take one step further. Distinguished from the fact that IC3/PDR and Forward CAR perform far better than Backward CAR on proving correctness in hardware verification [38, 41], Backward CAR performs well in verifying interlocking systems. It utilizes a different searching strategy compared to IC3/PDR and Forward CAR within the verification process. The hypothesis is the model structure of an interlocking system may be verified more efficiently by Backward CAR. This observation helps reveal the characteristic of interlocking verification and is worth further investigation.

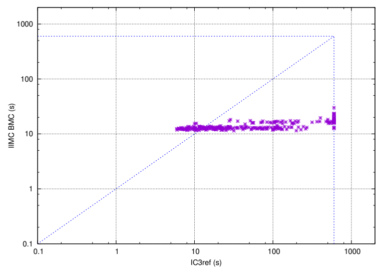
6 EXPERIENCE & LESSON

We initiated the design and development of the LightF3 framework in 2020, and over the course of the past three years, we have gained valuable experience and lessons, which we share in this section.

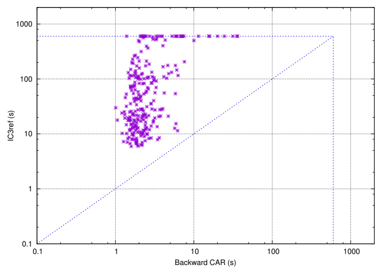
One of the most significant lessons we have learned is the importance of domain knowledge over specific implementation. We dedicated a substantial amount of time to consulting technicians and experts in the field to acquire deep domain knowledge of interlocking systems. This knowledge proved crucial in designing and developing the property instantiation procedure. When instantiating each function within a FQLTL property, we relied on the guidance of technicians to understand the meaning and specific requirements of the function. For example, when instantiating the function *BelongToTrack(switch, track)*, we consult technicians to understand its meaning (which is typically checking whether a specific switch belongs to a particular track) and how we determine its value for a given switch and track. Technicians may guide us to



(a) B-CAR vs IIMC-bmc

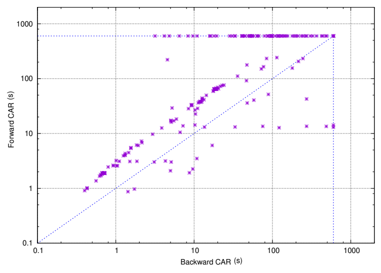


(b) IC3-ref vs IIMC-bmc

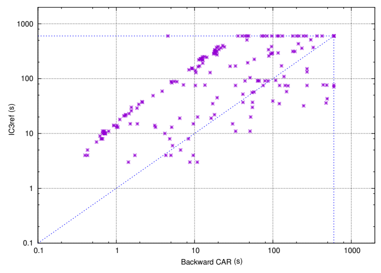


(c) B-CAR vs IC3-ref

Figure 9: Pairwise Comparison in Unsafe Concrete Property Level (within Station Eve)



(a) B-CAR vs F-CAR



(b) B-CAR vs IC3-ref

Figure 10: Pairwise Comparison in Abstract Property Level (within Station Eve)

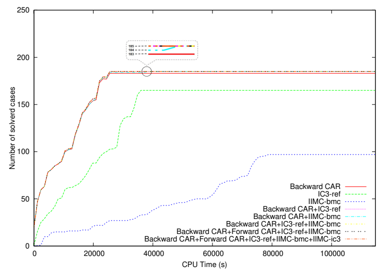


Figure 11: Sum of Cases Solved by Verfier Portfolios

refer to specific Excel sheets in the configuration data and verify whether the switch is located in a specific cell. This information then guides our code implementation.

Another important experience is that the best model-checking algorithm extensively benchmarked in the hardware-design domain may not be the most suitable one for interlocking system verification. Although IC3/PDR is considered the most advanced model checking algorithm in terms of overall performance, and its performance on the HWMCC benchmark is much better than Backward CAR, Backward CAR surprisingly outperforms IC3/PDR and others on the interlocking system verification benchmark. This could be because the interlocking system verification benchmark is unique, or because Backward CAR utilizes a different searching strategy compared to IC3/PDR. Therefore, it is always necessary and beneficial to try using a model checking portfolio, instead of only relying on the best algorithm by default.

7 DISCUSSION & CONCLUSION

Considerable efforts have been dedicated to modeling and verifying interlocking systems. However, there is a lack of formal descriptions for these systems, resulting in different perspectives and a lack of common understanding. In this paper, we address this gap by providing a formal description of the interlocking system and proposing a specific modeling and verification language, RIS-FL, based on the FQLTL logic.

Previous works have integrated various model-checking algorithms into their frameworks. However, these integrations often form closed chains that are challenging to extend. While model checking algorithms have rapidly advanced in recent decades, previous works have struggled to keep up with the state-of-the-art verification techniques. To address this issue, our framework, LightF3, is designed to be lightweight and extensible. It allows for the utilization of any new model checking improvements that support the aiger input format, ensuring that the latest verification techniques can be readily applied to interlocking system verification.

To conclude, we present LightF3, a lightweight and fully-process formal framework to model and verify Railway interlocking systems. A formal language RIS-FL based on FQLTL is provided for modeling the system and specifications. The RIS-FL models are automatically transformed into aiger models, enabling the invocation of third-party checkers to perform the verification task. To assess the effectiveness and efficiency of LightF3, we conduct evaluations on five real station instances obtained from our industrial partner. We further investigate and analyze the statistics of the verification results using various model-checking techniques. Overall, LightF3 offers a powerful and practical solution for modeling and verifying railway interlocking systems, bridging the gap between formal methods and railway domain expertise.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their helpful comments. This work is supported by the National Natural Science Foundation of China (NO. U21B2015 and 62002118), and the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

REFERENCES

[1] [n. d.]. IC3Ref. https://github.com/arbrad/IC3ref.

[2] Bowen Alpern and Fred B Schneider. 1987. Recognizing safety and liveness. *Distributed computing* 2, 3 (1987), 117–126. https://doi.org/10.1007/bf01782772

[3] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. 1999. Météor: A Successful Application of B in a Large Project. In *FM'99 — Formal Methods*, Jeannette M. Wing, Jim Woodcock, and Jim Davies (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–387.

[4] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[5] Armin Biere. 2007. The AIGER and-inverter graph (AIG) format version 20071012. (2007).

[6] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 317–320.

[7] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. [n. d.]. Bounded model checking. *Handbook of satisfiability* 185, 99 ([n. d.]), 457–481.

[8] Andrea Bonacchi, Alessandro Fantechi, Stefano Bacherini, Matteo Tempestini, and Leonardo Cipriani. 2014. Validation of Railway Interlocking Systems by Formal Verification, A Case Study. In *Software Engineering and Formal Methods*, Steve Counsell and Manuel Núñez (Eds.). Springer International Publishing, Cham, 237–252.

[9] Arne Borälv. 2018. Interlocking Design Automation Using Prover Trident. In *Formal Methods*, Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink (Eds.). Springer International Publishing, Cham, 653–656.

[10] Aaron R Bradley. 2011. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 70–87.

[11] Simon Busard, Quentin Cappart, Christophe Limbrée, Charles Pecheur, and Pierre Schaus. 2015. Verification of railway interlocking systems. In *ESSS*.

[12] Quentin Cappart, Christophe Limbrée, Pierre Schaus, and Axel Legay. 2015. Verification by discrete simulation of interlocking systems. In *29th Annual European Simulation and Modelling Conference*. 402–409.

[13] Quentin et al. Cappart. 2017. Verification of Interlocking Systems Using Statistical Model Checking. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. 61–68. https://doi.org/10.1109/HASE.2017.10

[14] Basri Tugcan Celebi and Ozgur Turay Kaymakci. 2016. Verifying the accuracy of interlocking tables for railway signalling systems using abstract state machines. *Journal of Modern Transportation* 24 (2016), 277–283.

[15] Yu Chen, Xiaoyu Zhang, and Jianwen Li. 2022. Finite Quantified Linear Temporal Logic and Its Satisfiability Checking. In *Artificial Intelligence Logic and Applications: The 2nd International Conference, AILA 2022, Shanghai, China, August 26–28, 2022, Proceedings*. Springer, 3–18.

[16] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. 2000. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 410–425.

[17] Alessandro Cimatti and Alberto Griggio. 2012. Software model checking via IC3. In *International Conference on Computer Aided Verification*. Springer, 277–293.

[18] Edmund M Clarke. 1997. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 54–56.

[19] Dalay Israel de Almeida Pereira, David Deharbe, Matthieu Perin, and Philippe Bon. 2019. B-Specification of Relay-Based Railway Interlocking Systems Based on the Propositional Logic of the System State Evolution. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky (Eds.). Springer International Publishing, Cham, 242–258.

[20] Niklas Eén, Alan Mishchenko, and Robert Brayton. 2011. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 125–134.

[21] BS EN. 2011. 50128 (2011). Railway Applications-Communication, Signalling and processing systems: Software for railway control and protection systems. *International Electrotechnical Commission* (2011).

[22] Alessio Ferrari, Gianluca Magnani, Daniele Grasso, and Alessandro Fantechi. 2011. Model checking interlocking control tables. In *FORMS/FORMAT 2010*. Springer, 107–115.

[23] Alessio Ferrari, Maurice H. ter Beek, Franco Mazzanti, Davide Basile, Alessandro Fantechi, Stefania Gnesi, Andrea Piattino, and Daniele Trentini. 2019. Survey on Formal Methods and Tools in Railways: The ASTRail Approach. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky (Eds.). Springer International Publishing, Cham, 226–241.

[24] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear temporal logic and linear dynamic logic on finite traces. *AAAI Press* (2013).

[25] Giuseppe De Giacomo and Moshe Y. Vardi. 2015. Synthesis for LTL and LDL on finite traces. *AAAI Press* (2015).

[26] Tim Gonschorek, Ludwig Bedau, and Frank Ortmeier. 2018. Bringing formal methods on the rail. *Safety and Reliability – Safe Societies in a Changing World* (2018).

[27] Anne E. Haxthausen, Jan Peleska, and Ralf Pinger. 2014. Applied Bounded Model Checking for Interlocking System Designs. In *Software Engineering and Formal Methods*, Steve Counsell and Manuel Núñez (Eds.). Springer International Publishing, Cham, 205–220.

[28] G. J. Holzmann. 1997. The Model Checker - SPIN. *IEEE Transactions on Software Engineering* 23 (1997), 279–295.

[29] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. 2013. The SafeCap Platform for Modelling Railway Safety and Capacity. In *Computer Safety, Reliability, and Security*, Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 130–137.

[30] Alexei Iliasov, Dominic Taylor, Linas Laibinis, and Alexander Romanovsky. 2018. Formal Verification of Signalling Programs with SafeCap. In *Computer Safety, Reliability, and Security*, Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 91–106.

[31] Alexei Iliasov, Dominic Taylor, Linas Laibinis, and Alexander Romanovsky. 2022. Practical Verification of Railway Signalling Programs. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–1. https://doi.org/10.1109/TDSC.2022.3141555

[32] Alexander Ivrii and Arie Gurfinkel. 2015. Pushing to the top. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 65–72.

[33] Phillip James, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger, Anton Setzer, Karim Kanso, and Simon Chadwick. 2014. Verification of Solid State Interlocking Programs. In *Software Engineering and Formal Methods*, Steve Counsell and Manuel Núñez (Eds.). Springer International Publishing, Cham, 253–268.

[34] Juan Bicarregui Jim Woodcock, Peter Gorm Larsen and John S. Fitzgerald. 2009. Formal methods: Practice and experience. *ACM Comput. Surv.* 41 (2009).

[35] Andrew Lawrence, Monika Seisenberger, Andrew Lawrence, and Monika Seisenberger. 2010. Verification of railway interlockings in scade. In *AVOCS'10, Proceedings of the 10th International Workshop on Automated Verification of Critical Systems and the Rodin User and Develop Workshop*. Springer, 112–114.

[36] Marie Le Bliguet and Andreas Andersen Kjær. 2008. *Modelling interlocking systems for railway stations*. Master's thesis. Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark.

[37] Michael Leuschel and Michael Butler. 2003. ProB: A model checker for B. In *International symposium of formal methods europe*. Springer, 855–874.

[38] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y Vardi. 2018. Simplecar: An efficient bug-finding tool based on approximate reachability. In *International Conference on Computer Aided Verification*. Springer, 37–44.

[39] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2013. LTL Satisfiability Checking Revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning (TIME)*.

[40] J. Li, S. Zhu, G. Pu, and M. Vardi. 2015. SAT-based Explicit LTL Reasoning. *Haifa Verification Conference* (2015).

[41] Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, and Moshe Y Vardi. 2017. Safety model checking with complementary approximations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 95–100.

[42] Bidhan Malakar and B.K. Roy. 2014. Railway fail-safe signalization and interlocking design based on automation Petri Net. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*. 1–4. https://doi.org/10.1109/ICICES.2014.7034154

[43] Ma Maofei and Zhang Yong. 2020. Modeling and Formal Verification of Interlocking System Based on UML and HCPN. In *2020 World Conference on Computing and Communication Technologies (WCCCT)*. 47–52. https://doi.org/10.1109/WCCCT49810.2020.9170006

[44] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*. Springer, 1–13.

[45] T. Michaud and M. Colange. 2018. Reactive synthesis from LTL specification with Spot. In *In Proceedings of the 7th Workshop on Synthesis*.

[46] Andrew Nash, Daniel Huerlimann, Jörg Schütte, and Vasco Paul Krauss. 2004. Railml† a standard data interface for railroad applications. *WIT Transactions on The Built Environment* 74 (2004).

[47] Kristin Y. Rozier and Moshe Y. Vardi. 2007. LTL Satisfiability Checking. In *International SPIN Workshop on Model Checking of Software*.

[48] Yakir Vizel and Arie Gurfinkel. 2014. Interpolating property directed reachability. In *International Conference on Computer Aided Verification*. Springer, 260–276.

[49] Linh Hong Vu, Anne E. Haxthausen, and Jan Peleska. 2017. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming* 133 (2017), 91–115. https://doi.org/10.1016/j.scico.2016.05.010 Formal Techniques for Safety-Critical Systems (FTSCS 2014).

[50] W. Zhu. 2021. Big Data on Linear Temporal Logic Formulas. In *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*.