https://doi.org/10.1007/s00165-021-00551-6

Formal Aspects of Computing (2021) 33: 925–955

Formal Aspects of Computing

Efficient data validation for geographical interlocking systems

Jan Peleska[1], Niklas Krafczyk[1,†], Anne E. Haxthausen[2] and Ralf Pinger[3]

[1] Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany
[2] DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark
[3] Siemens Mobility GmbH, Brunswick, Germany

Abstract. In this paper, an efficient approach to data validation of distributed geographical interlocking systems (IXLs) is presented. In the distributed IXL paradigm, track elements are controlled by local computers communicating with other control components over local and wide area networks. The overall control logic is distributed over these track-side computers and remote server computers that may even reside in one or more cloud server farms. Redundancy is introduced to ensure fail-safe behaviour, fault-tolerance, and to increase the availability of the overall system. To cope with the configuration-related complexity of such distributed IXLs, the software is designed according to the digital twin paradigm: physical track elements are associated with software objects implementing supervision and control for the element. The objects communicate with each other and with high-level IXL control components in the cloud over logical channels realised by distributed communication mechanisms. The objective of this article is to explain how configuration rules for this type of IXLs can be specified by temporal logic formulae interpreted on Kripke Structure representations of the IXL configuration. Violations of configuration rules can be specified using formulae from a well-defined subset of LTL. By decomposing the complete configuration model into sub-models corresponding to routes through the model, the LTL model checking problem can be transformed into a CTL checking problem for which highly efficient algorithms exist. Specialised rule violation queries that are hard to express in LTL can be simplified and checked faster by performing sub-model transformations adding auxiliary variables to the states of the underlying Kripke Structures. Further performance enhancements are achieved by checking each sub-model concurrently. The approach presented here has been implemented in a model checking tool which is applied by Siemens Mobility for data validation of geographical IXLs.

Keywords: Data validation, Interlocking systems, LTL, CTL, Model checking

Correspondence to: Jan Peleska, e-mail: peleska@uni-bremen.de
†Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project number 407708394.

1. Introduction

Background

Railway interlocking systems (IXLs) are designed according to different paradigms [Pac02, Chapter 4]. Two of the most widely used are (a) *route-based interlocking systems* and (b) *geographical interlocking systems*. The former are based on predefined routes through the rail network and use interlocking tables specifying safety conflicts between different routes and the point positions and signal states to be enforced before a route may be entered by a train. For design type (b), routes through the railway network can be allocated dynamically by indicating the starting and destination points of trains intending to traverse the railway network portion controlled by the IXL under consideration. In the original technology, electrical relay-based circuits were applied, whose elements and interconnections were designed in one-to-one correspondence with those of the physical track layout. The electric circuit design ensured dynamic identification of free routes from starting point to destination, the locking of points and setting of signals along the route, as well as on neighbouring track segments for the purpose of flank protection. In today's software-controlled electronic interlocking systems, instances of software components "mimic" the elements of the electric circuit, acting as *digital twins* of the associated physical track elements. Typically following the object-oriented paradigm, different components are developed, each corresponding to a specific type of physical track element, such as points, track sections associated with signals, and others with axle counters or similar devices detecting trains passing along the track. Similar to connections between electric circuit elements, instances of these software components are connected by communication channels reflecting the track network. The messages passed along these channels carry requests for route allocation, point switching and locking, signal settings, and the associated responses acknowledging or rejecting these requests. The software components are developed for re-use, so that novel interlocking software designs can be realised by means of configuration data, specifying which instances of software components are required, their attribute values, and how their communication channels shall be connected.

IXL design induces a distinguished verification and validation (V&V) step which is called *data validation*. For route-based IXLs, its main objective is to ensure completeness and correctness of interlocking tables. For geographical IXLs, the objective is to check whether the instantiation of software components is complete, each component is equipped with the correct attribute values, and whether the channel interconnections are adequate. Data validation becomes still more complex, if the IXL logic is distributed over track-side computers monitoring and controlling their associated physical track elements (local IXL logic) and cloud-based servers executing the global IXL logic. This approach is followed by Siemens Mobility with their new *Distributed Smart Safe System DS*^{3} which has been certified in 2020 [Son18, Pel20]. In addition to the digital twin configuration, aspects like deployment of software components, communication topology, and reconfiguration behaviour need to be configured for the software components residing in the cloud.

In any case, data validation objectives are specified by means of rules, and the rules collection is usually quite extensive (several hundred), so that manual data validation would be cumbersome, costly, and error-prone task. Also, manually programmed checking software is not a satisfactory solution, since the addition of new rules would require frequent extensions of the code. These extensions are costly, since data validation tools need to be validated according to tool class T2, as specified in the standard [CEN11]. Therefore, it is desirable to use data validation tools processing a logical query language to specify which rules should be enforced or which rule violations should be detected. This type of tool can be validated once and for all, since new validation rules can be specified by mans of new queries, without changing the software code.

Previous work

This paper is a follow-up contribution to [PKHP19], where the basic model checking principle for data validation of geographical IXLs has been presented. This principle was based on the following insights.

1. Exploiting known results about the temporal logic LTL, it has been shown that violations of safety-properties can be represented by a syntactic subset of LTL which is denoted by *data validation language (DVL)*. These considerations ensure that violations of IXL configuration rules can always be specified using this subset.
2. Exploiting known results about LTL and CTL, it was shown how LTL formulae ϕ representing safety violations (so-called *DVL-queries*) can be translated to CTL formulae $\Phi(\phi)$, such that CTL model checking of $\Phi(\phi)$ is

an *over-approximation* for LTL model checking of ϕ in the sense of abstract interpretation. This means that the absence of witnesses[1] for CTL formula $\Phi(\phi)$ implies the absence of solutions for LTL formula ϕ, which proves that no rule violations specified by ϕ are present.

3. For CTL, highly efficient and well-explored global model checking algorithms can be applied. These have complexity $O(\mid f \mid \cdot (\mid S \mid + \mid R \mid))$, where $\mid f \mid$ is the number of sub-formulae in CTL formula f, $\mid S \mid$ is the size of the state space, and $\mid R \mid$ is the size of the transition relation. Moreover, the application of CTL model checking is generally more efficient than LTL model checking, since the latter represents an NP-hard problem [CGP99, Section 4.2]. Explicit global model checking is an adequate approach to data validation, since the typical number of states (corresponding to track elements) to be expected is in the order of 10^6 for the largest IXL configurations.
4. A decomposition of the complete IXL configuration into sub-models corresponding to directed routes through the railway network allows for a significant speed-up of the checking process by processing sub-models concurrently.

To make this article self-contained, the essential parts of [PKHP19] have been reproduced here verbatim or with small additions.

Main contributions

In this article, the material presented in the previous work [PKHP19] is extended by the following contributions.

1. The underlying theory is presented here in a comprehensive form and with full proofs for the crucial lemmas and theorems involved.
2. The application of the theory to data validation is described in more detail and with additional examples that have not been published in [PKHP19]. In particular, examples concerning flank protection have been added.
3. The parallelisation concept used to speed up model checking is described in detail.
4. A solution for an unsolved problem stated in [PKHP19] is presented. As mentioned above, the application of CTL to sub-models instead of LTL results in an over-approximation which may lead to false alarms. We present a method and an associated algorithm, detecting false alarms by exploiting the finite LTL encoding elaborated in [BHJ+06].

It should be emphasised that the scientific contribution of [PKHP19] and of the present article consists in showing how *existing* knowledge about formal models, temporal logic, and model checking can be applied to solve a highly complex problem of the safety-critical systems domain, namely the automated data validation of IXL configurations. To our best knowledge, the approach presented here has never been proposed by other authors before, and no alternative industrial-strength data validation tool exists, possessing the same characteristics as the DVL-Checker presented here.

Overview

In Section 2, the data validation approach to geographical IXLs is explained from an engineering perspective. The mathematical foundations required to enable automated complete detection of IXL configuration rule violations are elaborated in Section 3. This is done without any reference to the intended application. The latter is described in Section 4, where the application of the mathematical theory to IXL data validation, including its parallelisation, is presented in detail. An algorithm for the detection of false alarms resulting from over-approximation is described and shown to be correct. Performance evaluation results are presented. Section 5 contains references to related work and competing approaches. Section 6 contains a conclusion.

2. Data validation for geographic interlocking systems

As indicated above, the software controlling geographical interlocking systems consists of objects communicating over channels, each instance representing a physical track element or a related hardware interface. The main types of track elements to be considered are points and diamond crossings, track segments, signals, and level crossings.

[1] A *witness* is a sequence of states fulfilling a temporal logic formula.

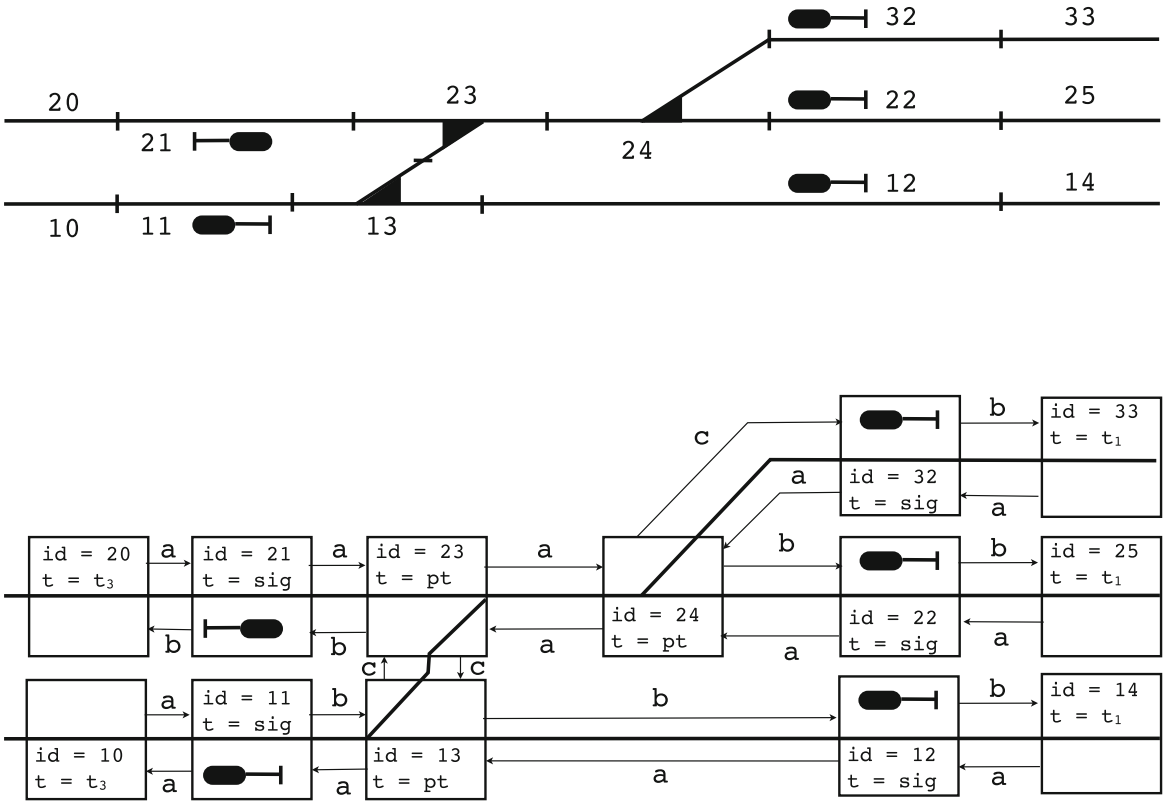


Fig. 1. Physical layout, associated software instances and channel connections.

The different tasks to be fulfilled by each track element at a specific position inside the track network require a large variety of sub-types, such as track segments acting as route interface elements or track segments acting as track vacancy detectors. Siemens structures the main types listed above into approximately 45 sub-types; and each track element sub-type is further specialised by a set of element-specific parameters that become attribute values of the objects they are represented by.

A subset of the channels—called *primary channels* in the following—reflect the physical interconnection between neighbouring track elements which are part of possible routes, to be dynamically allocated when a request for traversal from some starting point to a destination is given (Fig. 1). Other channels—called *secondary channels*—connect certain elements s_1 to others s_2, such that s_1 and s_2 are never neighbouring elements on a route, but s_2 may offer flank protection to s_1, when some route including s_1 should be allocated. Since geographical interlocking is based on request and response messages, each channel for sending request messages from some instance s_1 connected to an instance s_2 is associated with a "response channel" from s_2 to s_1. Primary channels are subsequently denoted by variable symbols a, b, c, d, while secondary channels are denoted by $e, f, g, \ldots$. Only points and diamond crossings use c-channels, and d-channels are used by diamond crossings only.

For signals, the driving direction they apply to is along channel a. For points, the straight track (point position "+") is always represented by the channel connections from a to b and vice versa, and the diverging track (point position "−") always from a to c and vice versa. The *stems* of a point are denoted by A, B, C according to the channels associated with the stem. The entry into/exit from the track network controlled by the interlocking systems is always marked by *border elements* of a special type. In Fig. 1, these types are denoted

by the fictitious identifiers t_1 and t_3. Some track sections may be crossed in both directions, so a border element may serve both as entry and exit element. This is discussed in more detail in the context of sub-model creation in Section 4.

All software instances are associated with a unique id and a type t corresponding to the track element type they are representing. Depending on the type, a list of further `int`-valued attributes $a_1, \ldots, a_k$ may be defined for each software instance. By using default value 0 for attributes that are not applicable to a certain component type, each element can be associated with the same complete list of attributes. Each valuation of a channel variable contains either a default value 0, meaning "no connection on this channel", or the instance identification $id > 0$ of the destination instance of the channel. Data validation rules state conditions about admissible sequences of element types and about admissible parameters.

In the following examples, when an element has the value n as its id, it is referred to as s_n.

Example 2.1 A typical pattern of data validation rules checks the existence of expected follow-up elements for an element of a given type.

Rule 1. From channel a of an element of type sig (i.e. a signal) pointing in downstream direction[2], an element of the same type with its a-channel also pointing downstream is found, before a border element of type t_1 or t_3 is reached.

Every rule can be transformed into a *rule violation condition*. For Rule 1, the violation would be specified as

Violation of Rule 1. From channel a of an element of type sig pointing in downstream direction, no element of the same type with its a-channel pointing downstream is found, before a border element of type t_1 or t_3 is reached.

The configuration in Fig. 1 violates Rule 1, because, for example, the path segment $\pi_1 = s_{21}.s_{23}.s_{24}.s_{22}.s_{25}$ contains the follow-up element s_{22}, but this is reached along π_1 via its a-channel. Practically, this means that the signal with id 22 does not point into the expected driving direction, so the expected route exit signal along π_1 is missing. An example of a path segment which is consistent with this rule is $\pi_2 = s_{32}.s_{24}.s_{23}.s_{13}.s_{11}.s_{10}$. □

Example 2.2 Another typical pattern of data validation rules refers to the element types that are required or admissible in certain segments of a route marked by elements of specific type.

Rule 2. From channel a of a signal of type sig pointing in downstream direction, there must be at least one element of type t_3, before the corresponding signal with type sig and channel a pointing in downstream direction is reached.

The corresponding rule violation can be specified as

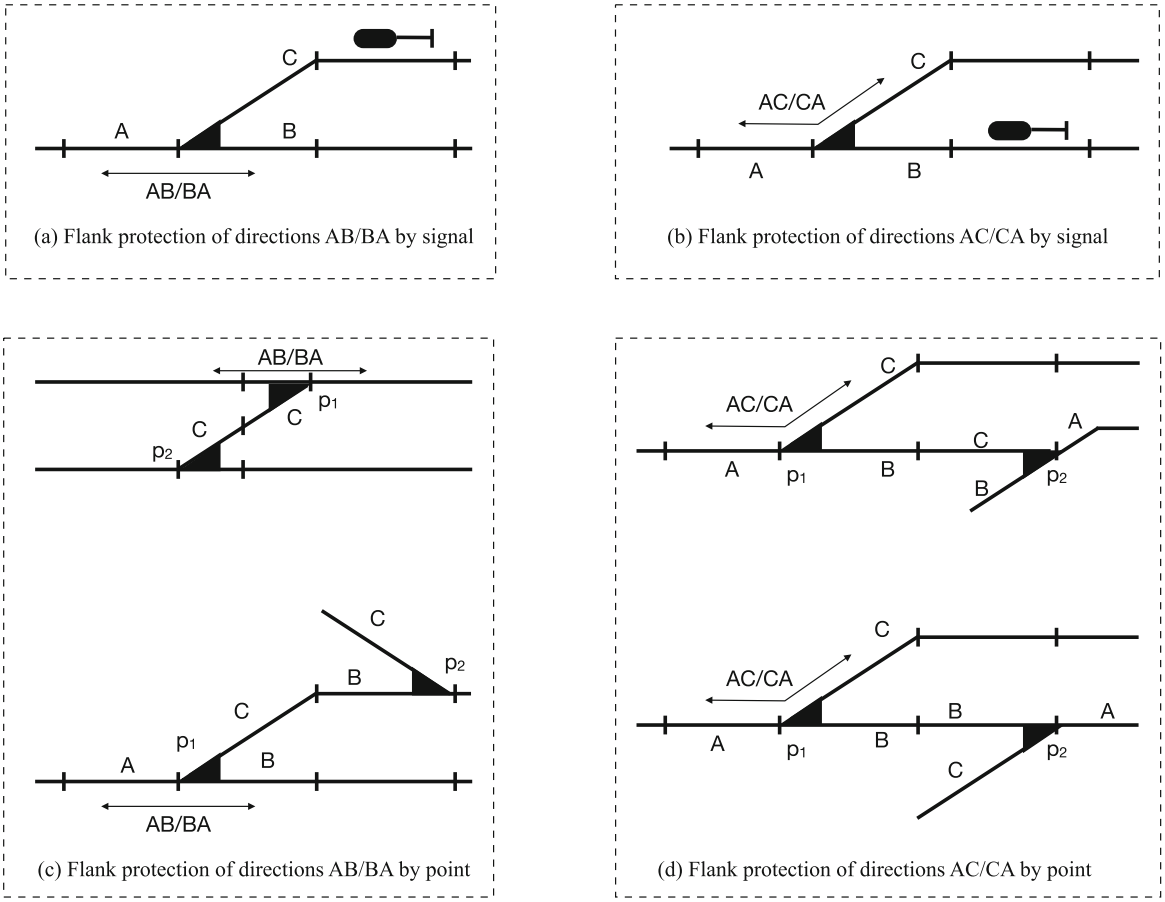
Violation of Rule 2. From channel a of a signal of type sig pointing in downstream direction, no element of type t_3 can be found before the corresponding signal with type sig and channel a pointing in downstream direction is reached.

The configuration in Fig. 1 violates this rule, because the path segments connecting the signals of type sig do not contain any element of type t_3. □

Example 2.3 Another typical pattern of data validation rules restricts the number of elements of a certain type that may be allocated between two elements of another type. The following fictitious rule illustrates this pattern (the real rules are slightly more complex and refer to other element types).

Rule 3. From channel a of a signal of type sig pointing in downstream direction, no more than k points ($t = pt$) are allowed, before the corresponding signal with type sig and channel a pointing in downstream direction is reached.

[2] This means that the signal is visible to trains driving in the direction of the a channel, see channel identifications for objects of type sig in Fig. 1. 'Downstream' denotes the driving direction.



(a) Flank protection of directions AB/BA by signal

(b) Flank protection of directions AC/CA by signal

(c) Flank protection of directions AB/BA by point

(d) Flank protection of directions AC/CA by point

Fig. 2. Several variants of flank protection.

The corresponding rule violation is specified as

Violation of Rule 3. From channel a of a signal of type sig pointing in downstream direction, more than k points ($t = pt$) are encountered, before the corresponding signal with type sig and channel a pointing in downstream direction is reached. □

Slightly more complex rules have to be specified for ensuring the correct configuration of elements offering flank protection to routes crossing points. In Fig. 2, several variants of signals and points offering flank protection to point p_1 are shown. Note that several more variants have to be considered in practise.

Flank protection by signal is shown for driving directions AB/BA in Fig. 2a and for driving directions AC/CA in Fig. 2b. Since flank protection by signal is unable to prevent collisions if the signals are disregarded, flank protection by point is the preferred solution, if available. Driving directions AB/BA of a point p_1 can be protected from trains entering the C-stem of p_1, if another point p_2 exists that may prevent trains from entering p_1's C-stem. This is illustrated in Fig. 2c. Driving directions AC/CA are protected from trains entering the B-stem of p_1 by points p_2 shown in Fig. 2d.

Example 2.4 The variants of flank protection shown in Fig. 2 lead to the following rules applicable to every element p_1 with type $t = pt$. It suffices to check flank protection for one driving direction, because then it also holds for the opposite driving direction. Therefore, the rules are only formulated for the case where the B and C-stems of the point under consideration point in driving direction.

Rule 4.1 (protection of driving direction AB/BA) If p_1's c-channel points in downstream direction, another point p_2 with its b channel or c-channel pointing towards the C-stem of p_1 is required, or a signal with a-channel pointing towards the C-stem of p_1 is required before another point p_3 with its a-channel pointing towards the C-stem of p_1 is encountered.
The condition about p_3 ensures that the flank protection is implemented not too far away from the point p_1 to be protected: after encountering a point like p_3, two signals instead of one would be required to protect p_1, because trains could approach p_1's C-stem via the B-stem or A-stem of p_3.

Rule 4.2 (protection of driving direction AC/CA) If p_1's b-channel points in downstream direction, another point p_2 with its b channel or c-channel pointing towards the B-stem of p_1 is required, or a signal with a-channel pointing towards the B-stem of p_1 is required before another point p_3 with its a-channel pointing towards the B-stem of p_1 is encountered.

For all points displayed in Fig. 1, Rule 4.1 and Rule 4.2 are fulfilled. The corresponding rule violations are specified as

Violation of Rule 4.1 If p_1's c-channel points in downstream direction, no other point p_2 with its b channel or c-channel pointing towards the C-stem of p_1 can be found, and no signal with a-channel pointing towards the C-stem of p_1 can be found before another point p_3 with its a-channel pointing towards the C-stem of p_1 is encountered, or a border element has been reached.

Violation of Rule 4.2 If p_1's b-channel points in downstream direction, no other point p_2 with its b channel or c-channel pointing towards the B-stem of p_1 can be found, and no signal with a-channel pointing towards the B-stem of p_1 can be found before another point p_3 with its a-channel pointing towards the B-stem of p_1 is encountered, or a border element has been reached. □

3. Logical foundations

3.1. Overview

In this section, the logical foundations of the model checking method for data validation are explained. The underlying theory is described without references to their practical application in the IXL context; the latter is explained in Section 4. The main results of this section are as follows.

1. The specification of rule violations that we use for data validation can be expressed by negations of LTL safety formulae (Section 3.4).
2. These negated formulae can always be expressed by LTL formulae using unquantified first-order formulae composed by path operators **X** (next) and **U** (until) only (Theorem 3.1).
3. Checking this type of LTL formulae can be performed by CTL model checking of transformed formulae: if the CTL check does *not* find a witness (a path) for the transformed formula, there is also none for the original LTL formula. This means that no rule violation exists (Section 3.6 and Theorem 3.3).
4. CTL checking is an over-approximation of LTL checking. As a consequence, *false alarms* may occur. These are witnesses for the transformed formula, but do not represent models of the original LTL formula. Since the manual verification or falsification of witness paths is cumbersome for users, an algorithm for the detection of false alarms is presented in the next section (Section 4.4).
5. The CTL model checking algorithms required for checking the formulae relevant for data validation are explained in Section 3.7.

In Section 4 it will be shown how IXL configurations may be interpreted as Kripke Structures, so that rule violations can be expressed in a natural way as negated LTL safety formulae over these configurations.

3.2. Kripke structures

A *State Transition System* is a triple $TS = (S, S_0, R)$, where S is the set of *states*, $S_0 \subseteq S$ is the set of *initial states*, $R \subseteq S \times S$ is the *transition relation*. The intuitive interpretation of R is that a state change from $s_1 \in S$ to $s_2 \in S$ is possible in TS if and only if $(s_1, s_2) \in R$.

Table 1. Expression evaluation.

$s(d) = d$	for integer constants d
$s(x\ \omega\ e) = s(x)\ \omega\ s(e)$	for variables x and expressions e and arithmetic operators $\omega \in \{+, -, /, *, <<, >>, \%\}$

Table 2. Semantics of atomic propositions.

$s \models \texttt{true}$
$s \not\models \texttt{false}$
$s \models v\ \nu\ d$ iff $s(v)\ \nu\ d$ for comparison operators $\nu \in \{=, \neq, <, \leq, >, \geq\}$
$s \models v\ \nu\ w$ iff $s(v)\ \nu\ s(w)$

A *Kripke Structure* $K = (S, S_0, R, L, AP)$ is a state transition system (S, S_0, R) augmented by a set AP of *atomic propositions* and a *labelling function* $L : S \longrightarrow 2^{AP}$ mapping each state s of K to the set of atomic propositions valid in s. Furthermore, it is required that the transition relation R is *total* in the sense that $\forall\, s \in S : \exists\, s' \in S : (s, s') \in R$.

A *computation* of a state transition system (or a Kripke structure) is an infinite sequence $\pi = s_0.s_1.s_2 \cdots \in S^\omega$ of states $s_i \in S$, such that the start state is an initial state, that is, $s_0 \in S_0$, and each pair of consecutive states is linked by the transition relation, that is, $\forall\, i > 0 : (s_{i-1}, s_i) \in R$. The terms *path* or *execution* are used synonymously for computations.

In the context of this paper, state spaces S consist of *valuation functions* $s : V \longrightarrow D$ mapping variable names from V to their actual values in D. For the context of this paper, it suffices to consider $D = \text{int}$, because all configuration parameters used for the interlocking systems under consideration may be encoded as integers. For the Boolean values `true`, `false`, the integer values 1, 0 are used, respectively.

3.3. First order formulae and their valuation

Given a Kripke Structure K with *variable valuation functions* $s : V \longrightarrow \text{int}$ as states, arithmetic expressions over variables from V are interpreted in a given state s by the rules shown in Table 1. These rules extend the domain of each valuation s to integer constants and arithmetic expressions over variables from V.

Atomic propositions are constructed by composing variables or arithmetic expressions using comparison operators. The valuation of atomic propositions is specified in Table 2, where d denotes integer constants, and v, w denote variables from V or arithmetic expressions over variables from V. We write $s \models p$ if p evaluates to `true` in state s, and $s \not\models p$ if p evaluates to `false`.

An *(unquantified) first-order formula* f over V is a logical formula with atomic propositions over V as specified above, composed by logical operators $\neg, \wedge, \vee$. The domain of valuation functions s is extended once more to first-order formulae, as specified in Table 3.

Table 3. Semantics of first-order formulae.

$s \models \neg f$ iff $s \not\models f$
$s \models f \wedge g$ iff $s \models f$ and $s \models g$
$s \models f \vee g$ iff $s \models f$ or $s \models g$

Table 4. Semantics of LTL formulae.

$\pi^i \models_{\text{LTL}} \texttt{true}$ for all $i \geq 0$
$\pi^i \not\models_{\text{LTL}} \texttt{false}$ for all $i \geq 0$
$\pi^i \models_{\text{LTL}} f$ iff $\pi(i) \models f$ if f is an unquantified first-order formula over V, to be evaluated as specified in Table 1, 2 and 3.
$\pi^i \models_{\text{LTL}} \neg\varphi$ iff $\pi^i \not\models_{\text{LTL}} \varphi$
$\pi^i \models_{\text{LTL}} \varphi \wedge \psi$ iff $\pi^i \models_{\text{LTL}} \varphi$ and $\pi^i \models_{\text{LTL}} \psi$
$\pi^i \models_{\text{LTL}} \varphi \vee \psi$ iff $\pi^i \models_{\text{LTL}} \varphi$ or $\pi^i \models_{\text{LTL}} \psi$
$\pi^i \models_{\text{LTL}} \mathbf{X}\varphi$ iff $\pi^{i+1} \models_{\text{LTL}} \varphi$
$\pi^i \models_{\text{LTL}} \mathbf{G}\varphi$ iff $\pi^{i+j} \models_{\text{LTL}} \varphi$ for all $j \geq 0$
$\pi^i \models_{\text{LTL}} \mathbf{F}\varphi$ iff there exists $j \geq 0$ such that $\pi^{i+j} \models_{\text{LTL}} \varphi$
$\pi^i \models_{\text{LTL}} \varphi\mathbf{U}\psi$ iff there exists $j \geq 0$ such that $\pi^{i+j} \models_{\text{LTL}} \psi$ and $\pi^{i+k} \models_{\text{LTL}} \varphi$ for all $0 \leq k < j$
$\pi^i \models_{\text{LTL}} \varphi\mathbf{W}\psi$ iff $\pi^{i+k} \models_{\text{LTL}} \varphi$ for all $k \geq 0$, or there exists $j \geq 0$ such that $\pi^{i+j} \models_{\text{LTL}} \psi$ and $\pi^{i+k} \models_{\text{LTL}} \varphi$ for all $0 \leq k < j$

3.4. Linear temporal logic LTL, safety properties and their violations

Linear temporal logic LTL

Linear Temporal Logic (LTL) is a logical formalism aiming at the specification of computation properties. The material presented here is based on [CGP99]. Given a Kripke structure with state valuations over variables from V, we use unquantified first-order LTL with the following syntax.

- Every unquantified first-order formula over V as specified above is an unquantified first-order LTL formula.
- If f, g are unquantified first-order LTL formulae, then $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}f$ (*Next*), $\mathbf{G}f$ (*Globally*), $\mathbf{F}f$ (*Finally*), $f\mathbf{U}g$ (*Until*), and $f\mathbf{W}g$ (*Weak Until*) are also unquantified first-order LTL formulae.

Operators $\mathbf{X}$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$, and $\mathbf{W}$ are called *path operators*.

The models of LTL formulae are infinite paths $\pi = s_0.s_1.s_2.\cdots \in S^\omega$; we write $\pi \models_{\text{LTL}} f$ if formula f holds on path π according to the semantic rules specified in Table 4.[3] We use notation $\pi^i = s_i.s_{i+1}.s_{i+2}\ldots$ to denote the path segment of π starting at element $\pi(i)$. A Kripke structure K fulfils LTL formula f if and only if every computation of K is a model of f:

$$K \models_{\text{LTL}} f \text{ iff } \pi \models_{\text{LTL}} f \text{ for all computations } \pi \text{ of } K$$

In the remainder of the paper, some equivalences between LTL formulae will be used in proofs. These are listed in the following lemma.

Lemma 3.1 Let φ, ψ be LTL formulae. Then

$$\begin{array}{rrr}
\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi) & \neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi & \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi \\
\mathbf{G}\varphi \equiv \varphi\ \mathbf{W}\ \texttt{false} & \mathbf{F}\varphi \equiv \neg\mathbf{G}\neg\varphi & \varphi\mathbf{U}\psi \equiv \varphi\mathbf{W}\psi \wedge \mathbf{F}\psi \\
\mathbf{F}\varphi \equiv \texttt{true}\mathbf{U}\varphi & \neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi & \neg\mathbf{G}\varphi \equiv \mathbf{F}\neg\varphi \\
\neg(\varphi\mathbf{W}\psi) \equiv \big(\neg\psi\mathbf{U}\neg(\varphi \vee \psi)\big) & &
\end{array}$$

[3] The operators $\vee$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$ are redundant and can be expressed using the remaining LTL operators alone. Therefore, they are sometimes introduced as syntactic abbreviations. For the purpose of this paper, however, it is better to represent their semantics in an explicit way.

Proof. We prove $\neg(\varphi \mathbf{W} \psi) \equiv \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big)$, since this equivalence is usually not to be found in standard text books, but is essential for our further considerations. The derivation is performed by transforming the left-hand side and right-hand side into their first-order representation and proving semantic equivalence of the latter. The other statements are established in an analogous way.

$$
\begin{aligned}
&\pi^i \models_{\mathrm{LTL}} \neg(\varphi \mathbf{W} \psi) \\
&\Leftrightarrow \pi^i \not\models_{\mathrm{LTL}} \varphi \mathbf{W} \psi \\
&\quad \text{[Semantics of } \neg\text{, Table 4]} \\
&\Leftrightarrow \neg\big(\forall\, k \geq 0 : \pi^{i+k} \models_{\mathrm{LTL}} \varphi\big) \wedge \neg\big(\exists\, j \geq 0 : (\pi^{i+j} \models_{\mathrm{LTL}} \psi \wedge \forall\, 0 \leq k < j : \pi^{i+k} \models_{\mathrm{LTL}} \varphi)\big) \\
&\quad \text{[Semantics of } \mathbf{W} \text{ (Table 4), negated]} \\
&\Leftrightarrow \big(\exists\, h \geq 0 : \pi^{i+h} \not\models_{\mathrm{LTL}} \varphi\big) \wedge \\
&\qquad \big(\forall\, j \geq 0 : (\pi^{i+j} \not\models_{\mathrm{LTL}} \psi \vee \exists\, 0 \leq k < j : \pi^{i+k} \not\models_{\mathrm{LTL}} \varphi)\big) \\
&\quad \text{[First-order logic rules for negation and quantification]} \\
&\Leftrightarrow \big(\exists\, h \geq 0 : \pi^{i+h} \models_{\mathrm{LTL}} \neg\varphi\big) \wedge \\
&\qquad \big(\forall\, j \geq 0 : (\pi^{i+j} \models_{\mathrm{LTL}} \neg\psi \vee \exists\, 0 \leq k < j : \pi^{i+k} \models_{\mathrm{LTL}} \neg\varphi)\big) \\
&\quad \text{[LTL semantics of } \neg\text{, (Table 4)]} \\
&\Leftrightarrow \big((\exists\, h \geq 0 : \pi^{i+h} \models_{\mathrm{LTL}} \neg\varphi) \wedge (\forall\, j \geq 0 : \pi^{i+j} \models_{\mathrm{LTL}} \neg\psi)\big) \vee \\
&\qquad \big(\exists\, j > 0 : (\pi^{i+j} \models_{\mathrm{LTL}} \psi \wedge \forall\ 0 \leq k < j : \pi^{k} \models_{\mathrm{LTL}} \neg\psi \wedge \exists\, 0 \leq h < j : \pi^{i+h} \models_{\mathrm{LTL}} \neg\varphi)\big) \\
&\quad \text{[First-order logic rules for } \vee, \wedge, \forall\text{, and } \exists\text{,} \\
&\qquad \text{note that second disjunct implies } \exists\, h \geq 0 : \pi^{i+h} \models_{\mathrm{LTL}} \neg\varphi\text{,} \\
&\qquad \text{note that } j \text{ must be greater zero, because otherwise } \pi^i \models_{\mathrm{LTL}} \varphi \mathbf{W} \psi\text{]} \\
&\Leftrightarrow \big(\exists\, h \geq 0 : (\pi^{i+h} \models_{\mathrm{LTL}} (\neg\varphi \wedge \neg\psi) \wedge \forall\, 0 \leq k < h : \pi^{i+k} \models_{\mathrm{LTL}} \neg\psi)\big) \\
&\quad \text{[First-order logic rules]} \\
&\Leftrightarrow \pi^i \models_{\mathrm{LTL}} \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big) \\
&\quad \text{[LTL semantics of } \mathbf{U}\text{, rules for } \wedge, \vee\text{]}
\end{aligned}
$$

□

Safety properties

A *safety property* P is a collection of computations $\pi \in S^\omega$, such that for every $\pi' \in S^\omega$ with $\pi' \notin P$, the fact that π' does *not* fulfil P can already be decided on a finite prefix of π'. It has been shown in [Sis94] that every safety property P can be characterised by a *Safety LTL* formula φ, so that the computations in P are exactly those fulfilling φ. The Safety LTL formulae are specified as follows [Sis94, Theorem 3.1]:

1. Every unquantified first-order formula is a Safety LTL-formula.
2. If φ, ψ are Safety LTL-Formulae, then so are

$$\varphi \wedge \psi, \quad \varphi \vee \psi, \quad \mathbf{X}\varphi, \quad \varphi \mathbf{W} \psi, \quad \mathbf{G}\varphi.$$

Observe that in these safety formulae, the negation operator must only occur in first-order sub-formulae.

Suppose that a safety property P is specified by Safety LTL formula φ. When looking for a path π *violating* φ, the violation $\pi \models_{\mathrm{LTL}} \neg\varphi$ can be equivalently expressed by a formula containing only first-order expressions composed by the operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$. This is shown in the following theorem.

Theorem 3.1 Let φ be a Safety LTL formula. Then *safety violation* $\neg\varphi$ can be equivalently expressed using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$.

Proof. We use structural induction over the syntax of safety LTL formulae.

Base case. If φ is a first-order expression, then its negation is again a first-order expression.

Induction hypothesis. Suppose that the negation of Safety LTL formulae φ, ψ can be expressed using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only.

Table 5. Interpretation of first-order expressions, conjunction and disjunction of LTL formulae.

$\mid [\varphi] \mid_i$	$0 \leq i \leq k$	
$\mid [f] \mid_i$	$s_i \models f$	f is an unquantified first-order expression
$\mid [\neg f] \mid_i$	$s_i \not\models f$	f is an unquantified first-order expression
$\mid [\psi_1 \wedge \psi_2] \mid_i$	$\mid [\psi_1] \mid_i \wedge \mid [\psi_2] \mid_i$	ψ_1, ψ_2 LTL formulae
$\mid [\psi_1 \vee \psi_2] \mid_i$	$\mid [\psi_1] \mid_i \vee \mid [\psi_2] \mid_i$	ψ_1, ψ_2 LTL formulae

Table 6. Interpretation rules for LTL path operators $\mathbf{X}$, $\mathbf{U}$ on acyclic paths.

$\mid [\varphi] \mid_i$	$0 \leq i < k$	i = k
$\mid [\mathbf{X}\psi] \mid_i$	$\mid [\psi] \mid_{i+1}$	`false`
$\mid [\psi_1 \mathbf{U} \psi_2] \mid_i$	$\mid [\psi_2] \mid_i \vee \big(\mid [\psi_1] \mid_i \wedge \mid [\psi_1 \mathbf{U} \psi_2] \mid_{i+1} \big)$	$\mid [\psi_2] \mid_i$

Induction step. Since every Safety LTL formula can be expressed using operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{W}$, $\mathbf{G}$, we need to show that the negations of $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{X}\varphi$, $\varphi \mathbf{W} \psi$, $\mathbf{G}\varphi$ can also be expressed using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$. To prove this, we use the equivalences for LTL formulae established in Lemma 3.1.

Case $\varphi \wedge \psi$. Since $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ and, according to the induction hypothesis, φ, ψ can be negated using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only, the induction step holds for operator $\wedge$.

Case $\varphi \vee \psi$. Since $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ and φ, ψ can be negated using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only, the induction step holds for operator $\vee$.

Case $\mathbf{X}\varphi$. Since $\neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$ and φ can be negated using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only, the induction step holds for operator $\mathbf{X}$.

Case $(\varphi \mathbf{W} \psi)$. Since $\neg(\varphi \mathbf{W} \psi) \equiv \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big) \equiv \big(\neg\psi \mathbf{U} (\neg\varphi \wedge \neg\psi)\big)$ and φ, ψ can be negated using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only, the induction step holds for operator $\mathbf{W}$.

Case $\mathbf{G}\varphi$. Since $\neg\mathbf{G}\varphi \equiv \mathbf{F}\neg\varphi \equiv (\mathtt{true}\mathbf{U}\neg\varphi)$ and φ can be negated using first-order expressions composed by operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only, the induction step holds for operator $\mathbf{G}$. This completes the proof. □

As a consequence of Theorem 3.1, a model checker specialised on the detection of safety violations only needs to support the evaluation of first-order formulae and operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$.

3.4.1. Safety violation formulae on finite paths

It will be explained in Section 4 how IXL configurations may be interpreted as Kripke structures K. Concrete model checking for uncovering rule violations will be performed on Kripke sub-models of K, whose transitions graphs are acyclic. This interpretation needs one relaxation of the Kripke structure definition $K = (S, S_0, R, L, AP)$: we admit state transition systems (S, S_0, R) whose transition relations are no longer total. In particular, all sub-model computations are finite, which follows trivially from the fact that finite, acyclic graphs cannot possess infinite paths.

From Theorem 3.1 above we know that the LTL formulae we are interested in—these express safety violations—can be represented using operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ only. Tables 5 and 6 present a semantic interpretation for such formulae on finite paths. This interpretation is based on more general results presented in [BHJ+06]. Given a finite path $\pi = s_0.s_1 \ldots s_k$, $\mid [\varphi] \mid_i$ denotes the interpretation of LTL formula φ on the path segment $\pi^i = s_i.s_{i+1} \ldots s_k$, so $\pi \models_{\mathrm{LTL}} \varphi$ holds if and only if $\mid [\varphi] \mid_0$ evaluates to `true`. Table 5 contains the interpretation rules for first-order (sub-)formulae in unquantified first-order LTL formulae φ. First-order formula f holds in segment π^i if and only if f holds in the segment's first state s_i, and its negation holds if f does *not* hold in s_i (recall the interpretation rules from Tables 1, 2, and 3). Conjunction and disjunction of arbitrary LTL formulae are defined in Table 5 in the usual way by distributing the operators through $\mid [\cdot] \mid$.

Table 6 specifies the interpretation of the temporal operators $\mathbf{X}$, $\mathbf{U}$. Exploiting the assumption that our transition graphs are acyclic, the rules do not have to deal with situations where the last state s_k coincides with a previous state on the same path. This general case is handled in [BHJ+06], but not needed in our context. As a consequence, the rule for interpreting $\mid [\mathbf{X}\psi] \mid_i$ just states that $\mid [\mathbf{X}\psi] \mid_i =$ `false` for $i = k$, because no next state exists where ψ could be evaluated in. For $i < k$, the usual interpretation is chosen: $\mid [\mathbf{X}\psi] \mid_i$ evaluates to `true` if and only if ψ holds on the segment π^{i+1}. For the until operator, the right-hand side operand must hold if $i = k$, otherwise the formula evaluates to `false`. For $i < k$, the usual recursive interpretation is chosen: $\mid [\psi_1 \mathbf{U} \psi_2] \mid_i$ is `true` if and only if ψ_2 holds on segment π^i or, alternatively, ψ_1 holds on π^i and $\psi_1 \mathbf{U} \psi_2$ holds on segment π^{i+1}.

Theorem 3.2 If the transition relation R of a Kripke structure K can be represented by a finite, acyclic, directed graph, then the semantic extension of LTL to finite paths specified above coincides with the finite linear encodings for LTL semantics introduced in [BHJ+06] that is used for bounded LTL model checking.

Proof. As described above, our interpretation in Tables 5 and 6 differs from the linear encodings specified in [BHJ+06] only in the cases $i = k$ for operators **X** and **U**. These cases are specified by more general formulae in [BHJ+06] which can be simplified to `false` for the **X**-operator and to $\mid [\psi_2] \mid_i$ for the **U**-operator, if all paths are acyclic and, therefore, do not contain any *lasso states* [BHJ+06, Section 1.3] that need to be considered in the case of potential cycles. □

The semantic rule for the **U**-operator in Table 6 is recursive. For the use of this in proofs it is sometimes practical to use equivalent non-recursive representations. Since the paths to be considered have finite length k, it is trivial to see that

$$\mid [\psi_2] \mid_i \vee \big(\mid [\psi_1] \mid_i \wedge \mid [\psi_1 \mathbf{U} \psi_2] \mid_{i+1} \big) \equiv \exists\, 0 \leq j \leq k - i.\ \big(\pi^{i+j} \models_{\mathrm{LTL}} \psi_2 \wedge \forall\, 0 \leq \ell < j.\ \pi^{i+\ell} \models_{\mathrm{LTL}} \psi_1 \big) \quad (1)$$

3.5. Computation tree logic CTL

Syntax of CTL formulae.

While LTL formulae have computations of Kripke structures as models, CTL has trees of computations as models. As a consequence, two new *path quantifiers* are introduced in addition to the path operators already known from LTL: Quantifier **E** denotes existential path quantification, in the sense that *"there exists a path segment starting at the current node of the computation tree, such that the formula specified after* **E** *holds on this segment."* Quantifier **A** denotes universal path quantification, in the sense that *"on all path segments starting at the current node of the computation tree the formula specified after* **A** *holds."* The CTL syntax is defined by the following grammar, where f denotes unquantified first-order formulae as specified in Section 3.3, formulae ϕ are called *state formulae*, and formulae ψ are called *path formulae*.

CTL-formula ::= ϕ
$\phi ::= f \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{E}\,\psi \mid \mathbf{A}\,\psi$
$\psi ::= \mathbf{X}\,\phi \mid \phi\ \mathbf{U}\ \phi$

According to this grammar, the path operators **X**, **U** can never be prefixed by another temporal operator in CTL. The same holds for the other path quantors which may be expressed in **X** and **U** according to Lemma 3.1. Only pairs consisting of path quantifier and temporal operator can occur in a row.

Semantics of CTL formulae.

The semantics of CTL formulae is explained using a Kripke structure K, specific states s of K and paths π through the computation tree of K. We write

$K, s \models_{\mathrm{CTL}} \phi$ (s a state of K, ϕ a state formula)

to express that ϕ holds in state s of K.

Table 7. Semantics of CTL formulae.

$K, s \models_{\text{CTL}} f$	iff	$s \models f$ for any unquantified first-order formula f with "$\models$" as defined in Table 3
$K, s \models_{\text{CTL}} \neg\phi$	iff	$K, s \not\models_{\text{CTL}} \phi$
$K, s \models_{\text{CTL}} \phi_1 \vee \phi_2$	iff	$K, s \models_{\text{CTL}} \phi_1$ or $K, s \models_{\text{CTL}} \phi_2$
$K, s \models_{\text{CTL}} \phi_1 \wedge \phi_2$	iff	$K, s \models_{\text{CTL}} \phi_1$ and $K, s \models_{\text{CTL}} \phi_2$
$K, s \models_{\text{CTL}} \mathbf{E}\,\psi$	iff	there is a path π from s such that $K, \pi \models_{\text{CTL}} \psi$
$K, s \models_{\text{CTL}} \mathbf{A}\,\psi$	iff	on every path π from s holds $K, \pi \models_{\text{CTL}} \psi$
$K, \pi^i \models_{\text{CTL}} \mathbf{X}\,\phi$	iff	$K, \pi(i+1) \models_{\text{CTL}} \phi$
$K, \pi^i \models_{\text{CTL}} \phi_0 \mathbf{U} \phi_1$	iff	there exists $j \geq 0$ such that $K, \pi(i+j) \models_{\text{CTL}} \phi_1$ and $K, \pi(i+k) \models_{\text{CTL}} \phi_0$ for all $0 \leq k < j$

We write

$K, \pi \models_{\text{CTL}} \psi$ (π a computation of K, ψ a path formula)

to express that ψ holds along path π through K. For CTL formulae ϕ we say ϕ *holds in the Kripke model* K and write $K \models_{\text{CTL}} \phi$ if and only if $K, s_0 \models_{\text{CTL}} \phi$ holds in every initial state s_0 of K. While this is useful for asserting that desired properties are fulfilled when starting from any initial state of K, it is not appropriate when wanting find witnesses for formulae expressing *unwanted* properties, such as the violations of IXL rules discussed in this paper. If the unwanted property is expressed by state formula ϕ, the model checker should return `true` or 'ALARM' if and only if

$\exists\, s_0 \in S_0.\ K, s_0 \models_{\text{CTL}} \phi$

The semantics of CTL formulae is specified in Table 7, where f denotes unquantified first-order formulae, ϕ, ϕ_i denote state formulae, and ψ, ψ_j denote path formulae. First-order formulae are interpreted just as in LTL, as specified in Table 3.

3.6. Over-approximation of LTL safety violation formulae by CTL

Full LTL and CTL have different expressiveness, and neither one is able to express all formulae of the other with equivalent semantics [CGP99]. In this section, however, it will be shown that any safety violation specified by an LTL formula f on a path π can also be detected by applying CTL model checking to a translated formula $\Phi(f)$ on any Kripke structure K containing π as a computation. This is, however, an *over-approximation*, in the sense that witnesses for $\Phi(f)$ in K will not always correspond to "real" rule violations in the IXL configuration. This will be illustrated by examples, and it is explained why the choice of sub-models described in Section 4.2 significantly reduces the number of such false alarms. Moreover, an algorithm for identifying false alarms is presented in Section 4.4.

Recalling from Theorem 3.1 that any safety violation can be specified using first-order formulae and operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$, we specify a partial transformation function $\Phi : \text{LTL} \not\longrightarrow \text{CTL}$ as follows.

$$
\begin{aligned}
\Phi(f) &= f \text{ for all first-order expressions } f \\
\Phi(f \wedge g) &= \Phi(f) \wedge \Phi(g) \\
\Phi(f \vee g) &= \Phi(f) \vee \Phi(g) \\
\Phi(\mathbf{X} f) &= \mathbf{EX}(\Phi(f)) \\
\Phi(f \mathbf{U} g) &= \mathbf{E}(\Phi(f) \mathbf{U} \Phi(g))
\end{aligned}
$$

Observe that Φ maps every LTL formula in its domain to a CTL state formula, since first-order expressions are state-formulae, and any LTL formula starting with a temporal operator is prefixed under Φ with the existential path quantifier $\mathbf{E}$. With this transformation at hand, the following theorem states that the absence of witnesses for $\Phi(f)$ in K guarantees the absence of a rule violation f on π.

From now on, we focus on *finite, acyclic* computations and use the interpretation of LTL formulae on finite, acyclic paths as specified in Tables 5 and 6. While some of the theorems to be presented below do hold in a more general setting, we only need the version for finite, acyclic paths. Moreover, not having to distinguish between finite and infinite paths facilitates the proof structures of most of the theorems we need in the sequel.

Theorem 3.3 Let π be any finite, acyclic path and f an LTL formula specifying a safety violation on π. Let K be a Kripke structure over state space S containing π as a computation. Then

$$\pi \models_{\text{LTL}} f \quad \text{implies} \quad K, \pi(0) \models_{\text{CTL}} \Phi(f).$$

Proof. The proof uses structural induction over the syntax of LTL formulae representing safety violations. These are expressed by first-order formulae and operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ according to Theorem 3.1. Throughout the proof, let $k = \mid \pi \mid -1$ be the last valid index of $\pi = \pi(0) \ldots \pi(\mid \pi \mid -1)$ and $\pi^i = \pi(i).\pi(i+1).\pi(i+2) \ldots \pi(k)$ be an arbitrary path segment of π with $0 \leq i \leq k$.

Base case. Suppose that $\pi^i \models_{\text{LTL}} g$ for an arbitrary first-order expression g. According to the semantic rules of LTL specified in Table 5 for first-order expressions, this is equivalent to $\pi(i) \models g$, with "$\models$" specified in Table 3. Since π is a computation of K by assumption, π^i is a path segment of K. Since the evaluation rules for first-order expressions are the same in LTL and CTL, $K, \pi(i) \models_{\text{CTL}} g$ follows. This argument was independent on the value of $0 \leq i \leq k$. Therefore, we can conclude from $\pi = \pi^0$ that $\pi \models_{\text{LTL}} f$ implies $K, \pi(0) \models_{\text{CTL}} f$ for any first-order expression f, which concludes the base case.

Induction hypothesis. Suppose that $\pi^i \models_{\text{LTL}} f$ and $\pi^i \models_{\text{LTL}} g$ imply $K, \pi(i) \models_{\text{CTL}} \Phi(f)$ and $K, \pi(i) \models_{\text{CTL}} \Phi(g)$, respectively, for given LTL formulae f, g expressing safety violations and any path segment π^i with $0 \leq i \leq k$.

Induction step. Using the induction hypothesis, it has to be shown that $\pi^i \models_{\text{LTL}} f \wedge g$, $\pi^i \models_{\text{LTL}} f \vee g$, $\pi^i \models_{\text{LTL}} \mathbf{X}f$, and $\pi^i \models_{\text{LTL}} f\mathbf{U}g$ imply that $K, \pi(i) \models_{\text{CTL}} \Phi(f) \wedge \Phi(g)$, $K, \pi(i) \models_{\text{CTL}} \Phi(f) \vee \Phi(g)$, $K, \pi(i) \models_{\text{CTL}} \mathbf{EX}\Phi(f)$, and $K, \pi(i) \models_{\text{CTL}} \mathbf{E}(\Phi(f)\mathbf{U}\Phi(g))$, respectively.[4]

Case $\pi^i \models_{\text{LTL}} f \wedge g$. This case is equivalent to $\pi^i \models_{\text{LTL}} f$ and $\pi^i \models_{\text{LTL}} g$ according to the LTL semantics specified in Table 5. According to the induction hypothesis, this implies $K, \pi(i) \models_{\text{CTL}} \Phi(f)$ and $K, \pi(i) \models_{\text{CTL}} \Phi(g)$. According to the CTL semantics specified in Table 7, this is in turn equivalent to $K, \pi(i) \models_{\text{CTL}} \Phi(f) \wedge \Phi(g)$.

Case $\pi^i \models_{\text{LTL}} f \vee g$. This case is shown in analogy to the previous case.

Case $\pi^i \models_{\text{LTL}} \mathbf{X}f$. This case is equivalent to $i < k \wedge \pi^{i+1} \models_{\text{LTL}} f$ according to the LTL semantics specified in Table 6. According to the induction hypothesis, this implies $K, \pi(i+1) \models_{\text{CTL}} \Phi(f)$. From the definition of Φ we know that $\Phi(f)$ is a state formula. Therefore, $\mathbf{EX}\Phi(f)$ is again a CTL state formula. From the CTL semantics in Table 7 and from the fact that $K, \pi(i+1) \models_{\text{CTL}} \Phi(f)$ has been established, we can derive $K, \pi^i \models_{\text{CTL}} \mathbf{X}\Phi(f)$, and, therefore, $K, \pi(i) \models \mathbf{EX}\Phi(f)$.

Case $\pi^i \models_{\text{LTL}} f\mathbf{U}g$. This case is equivalent to

$$\exists\, 0 \leq j \leq k - i.\ \left(\pi^{i+j} \models_{\text{LTL}} g \wedge \forall\, 0 \leq \ell < j.\ \pi^{i+\ell} \models_{\text{LTL}} f\right)$$

according to the LTL semantics specified in Table 6 and (1). This implies

$$\exists\, 0 \leq j \leq k - i.\ \left(K, \pi(i+j) \models_{\text{CTL}} \Phi(g) \wedge \forall\, 0 \leq \ell < j.\ K, \pi(i+\ell) \models_{\text{CTL}} \Phi(f)\right) \qquad (*)$$

according to the induction hypothesis. Since $\Phi(f)$, $\Phi(g)$ are state formulae, $\Phi(f)\mathbf{U}\Phi(g)$ is a path formula, and the CTL semantics specified in Table 7 shows that (*) implies $K, \pi^i \models_{\text{CTL}} \Phi(f)\mathbf{U}\Phi(g)$. As a consequence, $K, \pi(i) \models_{\text{CTL}} \mathbf{E}(\Phi(f)\mathbf{U}\Phi(g))$ holds as well. This completes the induction step and the proof of Theorem 3.3. □

3.7. CTL model checking

Basic concept of classical CTL model checking.

The CTL model checking algorithm used for IXL data validation is based on the "classical" algorithm described in [CGP99, Chapter 4]. It is specialised, however, on the CTL syntax required for uncovering safety violations. From Theorem 3.1 and Theorem 3.3 we know that for this purpose, only unquantified first-order formulae and the CTL operators $\wedge$, $\vee$, $\mathbf{EX}$, $\mathbf{EU}$ need to be supported. The algorithm's main concepts are summarised as follows.

[4] Recall that we do not have to consider negation, since this only occurs inside first-order formulae.

- The CTL specification formula is decomposed into its (binary) syntax tree.
- Starting at the leaves of the syntax tree (the leaves represent unquantified first-order formulae), the algorithm processes a sequence of sub-formulae ϕ_i (these are always state formulae) in bottom-up manner. This is implemented by means of a recursive in-order traversal of the syntax tree.
- The goal of each processing step is to annotate all states $s \in S$ satisfying $s \models_{\mathrm{CTL}} \phi_i$ with the new sub-formula ϕ_i. To this end, an additional labelling function label : $S \longrightarrow \mathbb{P}(CTL)$ is used, mapping each state to the set of sub-formulae it fulfils.
- The algorithm stops when the last formula ϕ_i having been processed coincides with the specification ϕ.
- The result of the algorithm is the set $S_\phi = \{s \in S \mid \phi \in \mathrm{label}(s)\}$.
- The Kripke model (S, S_0, R, L, AP) satisfies ϕ if its initial states are a subset of S_ϕ. This is of less interest for us, since our formulae ϕ will always represent safety violations, so it needs to be ensured that *none* of the initial states fulfil ϕ. Therefore, we check whether S_ϕ contains *at least one* initial state s_0 satisfying ϕ, i.e. we investigate whether

$$\exists\, s_0 \in S_0.\ K, s_0 \models_{\mathrm{CTL}} \phi \quad \text{which is equivalent to} \quad S_0 \cap S_\phi \neq \varnothing$$

Overview over the algorithm.

In Fig. 3, the entry function of the recursive algorithm is shown. *checkCTL* returns $S_\phi = \{s \in S \mid \phi \in \mathrm{label}(s)\}$, which is the set of all states satisfying the given formula ϕ. It remains to check whether at least one initial state of the Kripke structure K is contained in S_ϕ. Function *checkCTL* initialises an auxiliary function label : $S \longrightarrow 2^{\mathrm{CTL}}$ by mapping each state to a set containing only the atomic proposition `true`, which is fulfilled by every state. Auxiliary function label is passed as an in-out-parameter to every procedure called from *checkCTL* and its sub-procedures. In each sub-procedure, the image of label is extended by adding new entries to the sets label(s) of formulae fulfilled by certain states s.

In Fig. 4, the main function *calcLabel* of the algorithm is shown. It traverses the syntax tree representation of the formula ϕ to be checked and calls recursively itself or special sub-procedures for processing sub-formulae.

For evaluating unquantified first-order expressions, procedure *calcLabelFO* is used (Fig. 5). For each state $s \in S$, the procedure evaluates the expression according to the semantic rules stated in Tables 1, 2, 3 and adds the formula to label(s), if $s \models \phi$ holds.

For evaluating conjunctions $\phi = \phi_0 \wedge \phi_1$, each operand is evaluated separately by recursive calls to *calcLabel*, after which $\phi_i \in \mathrm{label}(s)$ holds for every state s fulfilling ϕ_i. Then sub-procedure *calcLabelAND* (Fig. 6) is called and adds ϕ to label(s) for all states s where label(s) contains both ϕ_0 and ϕ_1. Disjunctions are evaluated analogously, using sub-procedure *calcLabelOR* from Fig. 7.

For formulae $\phi = \mathbf{EX}\phi_0$, a recursive call to *calcLabel* first labels all states satisfying the operand formula ϕ_0. (Note that according to the syntax rules of CTL, ϕ_0 must be a state formula.) Then sub-procedure *calcLabelEX* (Fig. 8) checks all states s fulfilling ϕ_0 and inserts $\mathbf{EX}\phi_0$ into label(s') for all predecessor states s' of s.

Finally, formulae $\phi = \mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ are processed by first labelling states satisfying the operand (state) formulae, in analogy to conjunction and disjunction. Next, sub-procedure *calcLabelEU* (Fig. 9) identifies all states $s \in T$ satisfying ϕ_1: these also fulfil $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ and are labelled accordingly. Then predecessors s' of elements $s \in T$ are investigated: if they fulfil ϕ_0, they also fulfil $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$, since their successor s does. New states s' fulfilling $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ are added to T, so that their predecessors will be examined as well. Processed states are removed from T, so that the procedure terminates when T is empty.

Complexity considerations.

Studying the algorithms below, it is easy to see that the running time for checking $K \models_{\mathrm{CTL}} \phi$ is $O(|\,\phi\,| \cdot (|\,S\,| + |\,R\,|))$, where $|\,\phi\,|$ is the number of sub-formulae in CTL formula ϕ, $|\,S\,|$ is the size of the state space, and $|\,R\,|$ is the size of the transition relation. This is a well-known result which is elaborated, for example, in [CGP99, Theorem 1]. As a consequence, the running time is affected by the model size in a linear way only, while model size may affect the running time of bounded model checking in an exponential way. The running time is also lower than using LTL model checking algorithms directly, since the latter are NP-hard [CGP99, Section 4.2].

CTL algorithms over-approximate existence proof for LTL safety violations.

The following theorem states that the CTL model checking algorithms presented here are fit to uncover safety violations f specified in LTL, because the CTL solution space for the transformed formula $\Phi(f)$ is an over-approximation of the LTL solution space of f. As stated above, we focus on finite, acyclic paths satisfying safety violations, though the next theorem also holds in a more general context.

Theorem 3.4 Let $\pi \in S^*$ be a finite, acyclic path and f an LTL formula specifying a safety violation on π. Let K be a Kripke structure over state space S containing π as a computation. Then $\pi \models_{\text{LTL}} f$ implies that function *checkCTL* finds a witness for $\Phi(f)$, in the sense that $\pi(0) \in \mathit{checkCTL}(K, \Phi(f))$.

Proof. The proof is performed by structural induction over the formula syntax. For the induction to be applicable, we show in fact a stronger statement than that of the theorem. It is shown that after termination of $\mathit{calcLabel}(K, \Phi(f), \text{label})$,

$$\forall\, i \in \{0, \ldots, \mid \pi \mid -1\}.\ \pi^i \models_{\text{LTL}} g \Rightarrow \Phi(g) \in \text{label}(\pi(i)) \tag{2}$$

holds for arbitrary sub-formulae g of f, including f itself. The statement of the theorem follows from (2) for the special case $i = 0$ and $g = f$, because $\Phi(f) \in \text{label}(\pi(0))$ implies that $\pi(0) \in \mathit{checkCTL}(K, \Phi(f))$, as can be seen from the specification of *checkCTL* in Fig. 3.

Base case. Let g be a first-order sub-formula of f and suppose that $\pi^i \models_{\text{LTL}} g$. Then LTL semantics (Table 5) implies that $\pi(i) \models g$ according to the semantics of first-order formulae from Tables 1, 2, 3. Since g is a sub-formula of f and a first-order expression, it is located in one of the $\Phi(f)$-formula tree's leaves in an untransformed representation, since $\Phi(g) = g$. When *calcLabel* is called with $\Phi(g) = g$ as formula parameter, the procedure branches into procedure *calcLabelFO*, see Fig. 5. There, all states s fulfilling $s \models g$ will be labelled with g; in particular, $\Phi(g) = g$ will be added to $\text{label}(\pi(i))$, since π is a computation of K, so all $\pi(i)$ are states of K. This shows the validity of (2) for the base case.

Induction hypothesis. Assume that Statement (2) holds for LTL sub-formulae ϕ_0, ϕ_1 of f.

Induction step. We need to show that under the assumption of the induction hypothesis, Statement (2) also holds for LTL sub-formula g of f, with g being of the form $\phi_0 \wedge \phi_1$, $\phi_0 \vee \phi_1$, $\mathbf{X}\phi_0$, and $(\phi_0 \mathbf{U} \phi_1)$. Theorem 3.1 implies that we do not have to show anything for negated formulae, since, by assumption of the theorem, f specifies a negated safety formula, so negation only occurs inside first-order expressions.

Case $g = \phi_0 \wedge \phi_1$. Since $\pi^i \models_{\text{LTL}} g$ by assumption, the LTL semantics implies $\pi^i \models_{\text{LTL}} \phi_0$ and $\pi^i \models_{\text{LTL}} \phi_1$ (Table 5). The CTL checking algorithm is applied to $\Phi(g) = \Phi(\phi_0) \wedge \Phi(\phi_1)$. In case of a conjunction, procedure *calcLabel* of the checking algorithm first labels all states satisfying $\Phi(\phi_0)$ and $\Phi(\phi_1)$, respectively. By the induction hypothesis, this leads to state $\pi(i)$ being labelled with both $\Phi(\phi_0)$ and $\Phi(\phi_1)$. Next, procedure *calcLabel* calls procedure *calcLabelAND* (Fig. 6). There, state $\pi(i)$ is labelled with $\Phi(f) = \Phi(\phi_0) \wedge \Phi(\phi_1)$, because this state is already labelled with $\Phi(\phi_0)$ and $\Phi(\phi_1)$, as was to be shown.

Case $g = \phi_0 \vee \phi_1$ is verified in analogy to the previous case.

Case $g = \mathbf{X}\phi_0$. Since $\pi^i \models_{\text{LTL}} g$ by assumption, the LTL semantics on finite paths (see Table 6) implies $i <\mid \pi \mid -1$ and $\pi^{i+1} \models_{\text{LTL}} \phi_0$. The CTL checking algorithm is applied to $\Phi(g) = \mathbf{EX}\Phi(\phi_0)$. For state formulae $\mathbf{EX}\Phi(\phi_0)$, procedure *calcLabel* first labels all states satisfying $\Phi(\phi_0)$. The induction hypothesis implies that $\Phi(\phi_0) \in \text{label}(\pi(i+1))$. Moreover, since π is a computation of K, the pair $(\pi(i), \pi(i+1))$ is contained in the transition relation R of K. Therefore, the if-condition in *calcLabelEX* (see Fig. 8) evaluates to `true` for states $s = \pi(i)$ and $s' = \pi(i+1)$, and formula $\Phi(g) = \mathbf{EX}\Phi(\phi_0)$ is added to $\text{label}(\pi(i))$, as was to be shown.

Case $g = \phi_0 \mathbf{U} \phi_1$. Since $\pi^i \models_{\text{LTL}} g$ by assumption, the LTL semantics on finite paths (see Table 6) implies the existence of some $0 \leq j$, such that $i + j <\mid \pi \mid$, $\pi^{i+j} \models_{\text{LTL}} \phi_1$, and $\pi^{i+\ell} \models_{\text{LTL}} \phi_0$ for $0 \leq \ell < j$. Since $\Phi(g) = \mathbf{E}(\Phi(\phi_0)\mathbf{U}\Phi(\phi_1))$, function *calcLabel* first labels all states satisfying $\Phi(\phi_0)$, and then all states satisfying $\Phi(\phi_1)$. The induction hypothesis yields $\Phi(\phi_1) \in \text{label}(\pi(i+j))$ and $\Phi(\phi_0) \in \text{label}(\pi(i+\ell))$ for all $0 \leq \ell < j$ as a result of these two steps. Next, *calcLabel* calls $\mathit{calcLabelEU}(K, \Phi(\phi_0), \Phi(\phi_1), \text{label})$ (see Fig. 9). Since $\Phi(\phi_1) \in \text{label}(\pi(i+j))$ when this procedure is called, state $\pi(i+j)$ is added to queue T during its initialisation in *calcLabelEU*. In the following loop, formula $\mathbf{E}(\Phi(\phi_0)\mathbf{U}\Phi(\phi_1))$ is added to $\text{label}(\pi(i+j))$. The pairs $(\pi(i), \pi(i+1))$, $(\pi(i+1), \pi(i+2)), \ldots, (\pi(i+j-1), \pi(i+j))$ are all contained in the transition relation of K because π is a computation of K.

```
function checkCTL(in (S, S_0, R, L, AP) : KripkeStructure; in φ : CTL) : ℙ(S)
begin
  label : S ⟶ 2^CTL;
  label := {s ↦ {true} | s ∈ S};
  calcLabel((S, S_0, R, L, AP), φ, label);
  checkCTL := {s ∈ S | φ ∈ label(s)};
end
```

Fig. 3. Main algorithm for CTL property checking against Kripke structures.

```
procedure calcLabel(in (S, S_0, R, L, AP) : KripkeStructure;
                    in φ : CTL;
                    inout label : S ⟶ 2^CTL)
begin
  if φ is a first-order formula then
    calcLabelFO((S, S_0, R, L, AP), φ, label);
  elseif φ = φ_0 ∧ φ_1 then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelAND((S, S_0, R, L, AP), φ_0, φ_1, label);
  elseif φ = φ_0 ∨ φ_1 then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelOR((S, S_0, R, L, AP), φ_0, φ_1, label);
  elseif φ = EXφ_0 then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabelEX((S, S_0, R, L, AP), φ_0, label);
  elseif φ = E(φ_0 U φ_1) then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelEU((S, S_0, R, L, AP), φ_0, φ_1, label);
  endif
end
```

Fig. 4. Label calculation—control algorithm driven by formula syntax.

Therefore, the states involved will all be analysed in condition **if** $\mathbf{E}(\phi_0 \mathbf{U} \phi_1) \notin \text{label}(u) \wedge \phi_0 \in \text{label}(u)$ in the inner loop of procedure *calcLabelEU*. The order of these analyses is $u = \pi(i + j - 1), \pi(i + j - 2) \ldots, \pi(i)$, because $\pi(i + j - m)$ is always a predecessor of $\pi(i + j - m + 1)$ in K's transition relation. Moreover, $\Phi(\phi_0)$ is contained in each set $\text{label}(\pi(i + \ell))$ for each $0 \leq \ell < j$. Consequently, all these states $\pi(i + \ell)$ are labelled with $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ as well. In particular, for $\ell = 0$, this yields $\mathbf{E}(\phi_0 \mathbf{U} \phi_1) \in \text{label}(\pi(i))$, as was to be shown. □

```
procedure calcLabelFO(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ : First-order formula;
                      inout label : S ⟶ 2^CTL)
begin
  foreach s ∈ S do
    if s ⊨ φ then
      label(s) := label(s) ∪ {φ};
    endif
  enddo
end
```

Fig. 5. Algorithm for labelling states with first-order formulae.

```
procedure calcLabelAND(in (S, S_0, R, L, AP) : KripkeStructure;
                       in φ_0 : CTL; in φ_1 : CTL;
                       inout label : S ⟶ 2^CTL)
begin
  foreach s ∈ S do
    if φ_0 ∈ label(s) ∧ φ_1 ∈ label(s) then
      label(s) := label(s) ∪ {φ_0 ∧ φ_1};
    endif
  enddo
end
```

Fig. 6. Algorithm for labelling states with ($\phi_0 \wedge \phi_1$) formulae.

```
procedure calcLabelOR(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ_0 : CTL; in φ_1 : CTL;
                      inout label : S ⟶ 2^CTL)
begin
  foreach s ∈ S do
    if φ_0 ∈ label(s) ∨ φ_1 ∈ label(s) then
      label(s) := label(s) ∪ {φ_0 ∨ φ_1};
    endif
  enddo
end
```

Fig. 7. Algorithm for labelling states with ($\phi_0 \vee \phi_1$) formulae.

4. Model checking of IXL configurations

This section explains how an IXL software configuration can be represented as a Kripke structure having a state for each element and a transition from one element to another element, whenever the first element has a channel with the second element as its destination.

```
procedure calcLabelEX(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ : CTL;
                      inout label : S ⟶ 2^CTL)
begin
  foreach s ∈ S do
    if ∃s' ∈ S : R(s, s') ∧ φ ∈ label(s') then
      label(s) := label(s) ∪ {EXφ};
    endif
  enddo
end
```

Fig. 8. Algorithm for labelling states with $\mathbf{EX}\phi$ formulae.

```
procedure calcLabelEU(in (S, S0, R, L, AP) : KripkeStructure;
                      in φ0 : CTL; in φ1 : CTL;
                      inout label : S ⟶ 2^CTL)
begin
  T := ⟨s ∈ S | φ1 ∈ label(s)⟩;
  foreach s ∈ T do
    label(s) := label(s) ∪ {E(φ0 U φ1)};
  enddo
  while T ≠ ⟨ ⟩ do
    s := head(T);
    T := tail(T);
    foreach u ∈ {v ∈ S | R(v, s)} do
      if E(φ0 U φ1) ∉ label(u) ∧ φ0 ∈ label(u) then
        label(u) := label(u) ∪ {E(φ0 U φ1)};
        T := T ⌢ ⟨u⟩;
      endif
    enddo
  enddo
end
```

Fig. 9. Algorithm for labelling states with $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ formulae.

4.1. IXL configurations as Kripke structures

The configurations for geographical IXLs described in Section 2 give rise to Kripke structures $K = (S, S_0, R, L, AP)$ with variable symbols from some set V as follows (symbol d denotes `int`-values).

$$
\begin{aligned}
V &= \{id, t\} \cup C \cup A \\
C &= \{c \mid c \text{ is a primary or secondary channel symbol}\} \\
A &= \{a \mid a \text{ is an attribute symbol}\} \\
S &= \{s : V \longrightarrow \text{int} \mid \text{There exists a configuration instance with} \\
&\qquad\qquad \text{id, type, channel, and attribute valuation } s\} \\
S_0 &= S \\
R &= \{(s, s') \mid \exists\, c \in C : s(c) = s'(id)\} \\
AP &= \{id = d \mid \exists\, s \in S : s(id) = d\} \cup \{t = d \mid \exists\, s \in S : s(t) = d\} \cup \\
&\quad \{c = d \mid c \in C \wedge \exists\, s \in S : s(c) = d\} \cup \{a = d \mid a \in A \wedge \exists\, s \in S : s(a) = d\} \\
L &: S \longrightarrow 2^{AP};\ \ s \mapsto \{v = d \mid v \in V \wedge s(v) = d\}
\end{aligned}
$$

Each K-state in S is represented by a valuation function s mapping id, type, channel, and attribute symbols to corresponding integer values, such that there is a configuration element with exactly these values. The atomic propositions consist of all equalities $v = d$, where v is a symbol of V and d an integer value occurring for v in at least one configuration element. Every K-state is an initial state, because configuration rules are checked from any element as starting point. Two elements s, s' are linked by the transition relation whenever s has a channel c connected to s'; this is expressed by $s(c)$ carrying the id of s'. The labelling function maps each state s exactly to the propositions $v = s(v),\ v \in V$ that are valid in this state. Using the state valuation rules specified in Section 3.3, this can be equivalently expressed by $L(s) = \{v = d \mid s \models v = d\}$.

The *transition graph* of K is a directed graph (S, R) with K-states S as its set of nodes and K's transition relation R as its set of edges. Each edge (s, s') is labelled with channel symbol $c \in C$ if and only if $s(c) = s'(id)$, that is, if and only if a c-channel emanting from s ends at s'.

With the Kripke structure at hand, IXL configuration rules can be expressed by LTL Safety formulae, so rule violations may be expressed in LTL using first-order formulae and operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$, as shown in Section 3. Specifying rule violations on Kripke structure K representing a complete IXL configuration is quite complicated, however, because most rules refer to routes traversed in a certain driving direction, whereas K's transition relation connects any pair of configuration elements linked by any channel. This results in computations that do not correspond to any "real" route through the network.

Example 4.1 The Kripke structure corresponding to the configuration shown in Fig. 1 has a finite path

$$s_{10}.s_{11}.s_{13}.s_{23}.s_{21}.s_{20},$$

because all elements in this sequence are linked by some channel a, b, c. This path, however, cannot be realised as a train route, due to the topology of points s_{13} and s_{23}. □

In [HPP13], this problem has been overcome by using existentially quantified LTL with rigid variables as introduced in [MP92]. Apart from the fact that quantified LTL formulae are harder to create and understand, this would not allow for the over-approximation by means of CTL as described in Section 3.6. Therefore, we will now introduce sub-models of full configuration models where the problem of infeasible paths no longer occurs.

4.2. Sub-models

The *border elements* of an IXL configuration can be identified by the fact that only one of the main channels a, b is connected to another element, while the other channel is undefined. Element 20 in Fig. 1, for example, is a border element, because it has channel a connected to element 21, while channel b remains unconnected. Points or diamond crossings are never used as border elements, so only channels a, b need to be considered when identifying border elements in the Kripke structure K representing the complete configuration. Each border element introduces a well-defined driving direction specified by the channel which is defined and, therefore, "points into" the network specified by the configuration.

A sub-model is now created for every border element s_{bdr} as a Kripke structure $K(s_{bdr})$ which is a sub-structure of Kripke structure K representing the whole IXL configuration, as described above in Section 4.1. A sub-model is created according to the following rules.

1. The *driving direction* associated with $K(s_{bdr})$ corresponds to the direction specified by the defined channel a or b of border element s_{bdr}.
2. The transition graph of $K(s_{bdr})$ is the largest rooted, acyclic, directed sub-graph G of K's transition graph, such that the following properties hold.
   (a) G has root node s_{bdr}.
   (b) Each node which is reachable via edges in driving direction is part of G.
   (c) For nodes s representing points entered by their B-stem or C-stem, the only continuation is via the element s' connected to the points' A-stem. This means that edge (s, s') is labelled by an a channel.
   (d) For nodes s representing points entered by their A-stem, the continuations are via the elements s' connected to the points' B-stem or C-stem. This means that edge (s, s') is labelled by a b or c channel.
   (e) For diamond crossings entered via A,B,C,D-stem, the only possible continuations are via elements connected to the D,C,B,A-stems, respectively.
   (f) The graph expansion stops at node $s1$ and edge $(s1, s2)$, if $s2$ is already contained in the set of nodes. In this case, $(s1, s2)$ is not added to the edges of the sub-model.
   (g) The graph expansion stops when a node s represents a track element which is reached by its defined channel, so that no outgoing channel is available. In other words, the node s which has been reached is another border element.
3. The states of $K(s_{bdr})$ are the nodes of G.
4. Every state of $K(s_{bdr})$ is an initial state.
5. Every element of the sub-model is equipped with additional Boolean (i.e. {0, 1}-valued) attributes $dirA$, $dirB$, $dirC$, $dirD$ with value 1 if its respective channel a, b, c, or d points in driving direction; otherwise the attribute carries value 0. Note that for points and diamond crossings, several $dirX$-attributes can have value 1.
6. Every element is associated with Boolean attributes upA, upB, upC, upD ("upstream A, B, C, D"). For a given element s in a sub-model, $upA = 1$ if and only if there exists a predecessor element s' which is linked by its a-channel to s. For B, C, D, the attribute values are analogously defined.
7. Further auxiliary attributes are added to each sub-model state as described in Section 4.3 below.

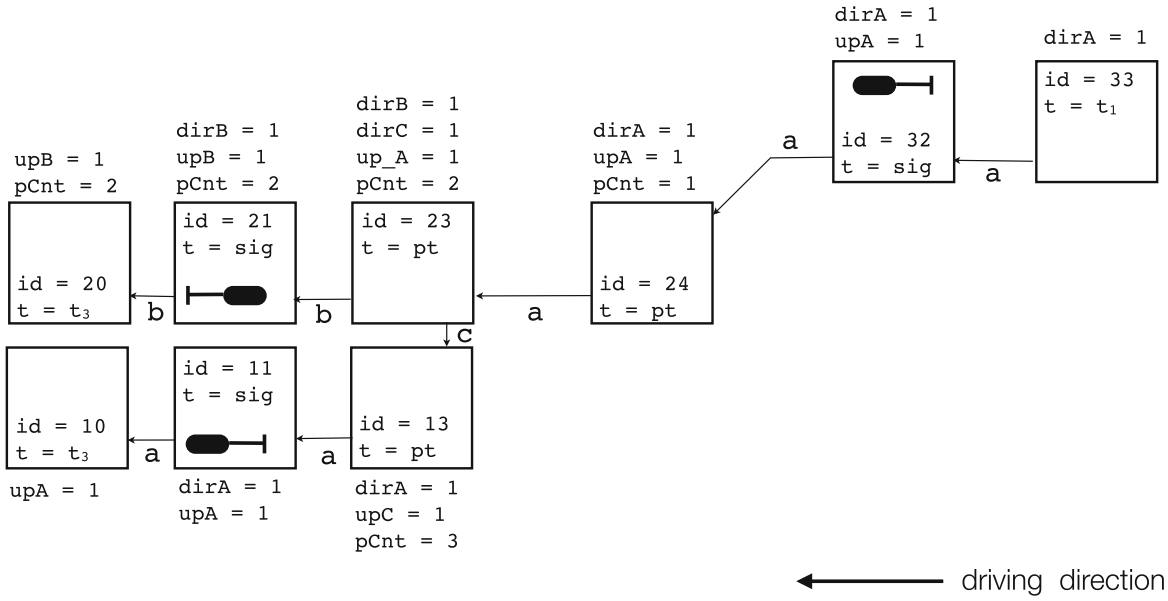


Fig. 10. Sub-model created from border element s_{33} in Fig. 1. Only attributes with positive value are shown.

The sub-model creation procedure described above is performed by means of a *depth-first search* on the transition graph (S, R) of the full model K. Therefore, the running time for the creation of one sub-model is $O(\mid S \mid + \mid R \mid)$ [CLRS09, p. 606]. The number of sub-models to be created equals the number of border elements contained in the IXL configuration.

Example 4.2 The complete IXL configuration depicted in Fig. 1 has border elements s_{10}, s_{20}, s_{33}, s_{25}, s_{14}. The sub-model resulting from border element s_{33} is shown in Fig. 10, together with the new auxiliary attributes $dirA, \ldots$ (the meaning of attribute $pCnt$ is explained in Section 4.3 below). Element s_{33} induces the driving direction along its channel a; since it is a border element, its channel b is not linked to another element. □

4.3. Specifying rule violations on sub-models

As indicated before, configuration rules for IXL configurations can always be represented by LTL safety conditions to be fulfilled by all paths of all sub-models: every *violation* of a configuration rule can be decided on a finite path of track element configurations, where consecutive elements on the path are linked by primary channels. Consequently, when checking for rule violations ϕ, each of these formulae ϕ is a negated LTL safety formula, and therefore described by means of first-order expressions and operators $\vee$, $\wedge$, $\mathbf{X}$, $\mathbf{U}$, as explained in the previous section.

The description of rule violations in LTL becomes rather straightforward when specified for sub-models; this is illustrated in the following examples.

Example 4.3 The rule violation specified in Example 2.1, when applied to a sub-model as the one depicted in Fig. 10, can be expressed in unquantified first-order LTL as

$$\phi_1 \equiv t = sig \wedge \mathrm{dirA} = 1 \wedge \mathbf{X}\big((t \neq sig \vee dirA = 0)\mathbf{U}(t = t_1 \vee t = t_3)\big)$$

This LTL formula is translated via Φ defined in Section 3.6 into CTL formula

$$\Phi(\phi_1) \equiv t = sig \wedge \mathrm{dirA} = 1 \wedge \mathbf{EX}\Big(\mathbf{E}\big((t \neq sig \vee dirA = 0)\mathbf{U}(t = t_1 \vee t = t_3)\big)\Big)$$

The only witness for $\Phi(\phi_1)$ in the sub-model shown on Fig. 10 is the path $s_{32}.s_{24}.s_{23}.s_{21}.s_{20}$, and this is also a witness for ϕ_1, so in this example, the CTL over-approximation does not produce any false alarms in this case. □

Example 4.4 The rule violation specified in Example 2.2, when applied to a sub-model, can be expressed in unquantified first-order LTL as

$$\phi_2 \equiv t = sig \wedge \mathrm{dirA} = 1 \wedge \mathbf{X}\big(t \neq t_3\mathbf{U}(t = sig \wedge \mathrm{dirA} = 1)\big)$$

This LTL formula is translated via Φ defined in Section 3.6 into CTL formula

$$\Phi(\phi_2) \equiv t = sig \wedge \mathrm{dirA} = 1 \wedge \mathbf{EX}\Big(\mathbf{E}\big(t \neq t_3\mathbf{U}(t = sig \wedge \mathrm{dirA} = 1)\big)\Big)$$

It is easy to see that for the sub-model shown in Fig. 10, the only witness is given by path $s_{32}.s_{24}.s_{23}.s_{13}.s_{11}$, so, again, no false alarms exist for this rule violation. □

The rule violations 4.1 and 4.2 associated with flank protection as described in Example 2.4 can be formalised as follows. In the formulae below, we use abbreviation

$$boundary \equiv (dirA + dirB + dirC + dirD = 0).$$

boundary evaluates to `true` if and only if the element is an exit border element, since they are the only ones without outgoing channels in driving direction.

Example 4.5 The rule violations 4.1 and 4.2 associated with flank protection as described in Example 2.4 can be formalised as follows.

$$\begin{aligned}\phi_{4.1} \equiv\ & t = pt \wedge \mathrm{dirC} = 1 \wedge \mathbf{X}\big(upC = 1 \wedge \\ & ((t \neq pt \vee dirA = 0) \wedge (t \neq sig \vee dirA = 1)) \\ & \quad \mathbf{U}(boundary \vee (t = pt \wedge dirA = 0))\big)\end{aligned}$$

Condition $\mathbf{X}upC = 1$ means that we are only interested in paths where the successor element of the point p_1 is connected to p_1's C-stem. The left operand of the **U**-operator specifies that no protecting points or signals are found. The right hand side of the **U**-operator specifies that we stop looking for suitable flank protection as soon as we have found a point offering no protection (this is equivalent to its a-channel pointing back towards p_1) or if the end of the route has been reached.

This LTL formula is translated via Φ defined in Section 3.6 into CTL formula

$$\begin{aligned}\Phi(\phi_{4.1}) \equiv\ & t = pt \wedge \mathrm{dirC} = 1 \wedge \mathbf{EX}\big(upC = 1 \wedge \\ & \mathbf{E}((t \neq pt \vee dirA = 0) \wedge (t \neq sig \vee dirA = 1)) \\ & \quad \mathbf{U}(boundary \vee (t = pt \wedge dirA = 0))\big)\end{aligned}$$

The formalisation of rule violation 4.2 (erroneous protection for driving directions AC/CA) is specified in LTL as follows. Example 2.4 can be formalised as follows.

$$\begin{aligned}\phi_{4.2} \equiv\ & t = pt \wedge \mathrm{dirB} = 1 \wedge \mathbf{X}\big(upB = 1 \wedge \\ & ((t \neq pt \vee dirA = 0) \wedge (t \neq sig \vee dirA = 1)) \\ & \quad \mathbf{U}(boundary \vee (t = pt \wedge dirA = 0))\big)\end{aligned}$$

and in translated form as

$$\Phi(\phi_{4.2}) \equiv t = pt \wedge \mathrm{dirB} = 1 \wedge \mathbf{EX}\big(upB = 1 \wedge \mathbf{E}((t \neq pt \vee dirA = 0) \wedge (t \neq sig \vee dirA = 1)) \mathbf{U}(boundary \vee (t = pt \wedge dirA = 0))\big)$$

□

We have seen above that auxiliary attributes can be introduced during sub-model creation, in order to facilitate the construction of rule violation formulae. Moreover, these attributes may be used to speed up the checking process.

Consider again the Example 2.3 in Section 2, where the number of elements of a certain type located between two reference elements needs to be counted. In principle, violation formulae associated with rules of that kind could be specified using *Counting LTL*, an extension of LTL allowing to check whether a path fulfils constraints referring to the number of states fulfilling certain properties [LMP10]. Checking Counting LTL formulae, however, is EXPSPACE-complete, and therefore, we cannot expect to find model checking algorithms for Counting LTL that are as efficient as the CTL-algorithms presented above.

Instead, a new auxiliary attribute $pCnt$ is introduced during sub-model creation. In every state of the sub-model, this attribute contains the number of points encountered in driving direction so far. This is illustrated in Fig. 10.

Example 4.6 With auxiliary attribute $pCnt$ at hand, the violation of Rule 3 from Example 2.3 is specified in LTL as

$$\phi_3 \equiv t = sig \wedge dirA = 1 \wedge \mathbf{X}\big((t \neq sig \vee dirA = 0)\ \mathbf{U}\ pCnt > k\big)$$

The operand of $\mathbf{X}$ expresses that until $pCnt > k$, no signal pointing in the same direction as the starting signal is met, i.e. if a signal is met, it must point in the opposite direction. Translated to CTL, this results in

$$\Phi(\phi_3) \equiv t = sig \wedge dirA = 1 \wedge \mathbf{EX}\Big(\mathbf{E}\big((t \neq sig \vee dirA = 0)\ \mathbf{U}\ pCnt > k\big)\Big)$$

Assuming that $k \geq 3$, there are obviously no witnesses for $\Phi(\phi_3)$ in the sub-model from Fig. 10. For $k = 2$, checking $\Phi(\phi_3)$ results in witness $s_{32}.s_{24}.s_{23}.s_{13}$, and again, this is also a witness for the LTL formula ϕ_3.

□

In analogy to the example shown here, further auxiliary attributes are added by the DVL Checker during sub-model creation.

4.4. Detecting false alarms

The following example shows how the CTL over-approximation for checking LTL formulae on non-linear models may lead to false alarms.

Example 4.7 Consider the transition graph of a Kripke structure $K(s_0)$ sketched in Fig. 11 with root node s_0 and atomic propositions p, q. It is fictitious, but this graph pattern might well occur in an IXL sub-model with driving direction $s_0 \longrightarrow s_1$, where node s_2 represents a point.

Each node in Fig. 11 is annotated with the propositions fulfilled in the corresponding Kripke-state. For example, s_1 satisfies p but not q, s_4 fulfils p and q, and s_3 satisfies neither p nor q.

Suppose we wish to prove the absence of a witness for LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$. Applying the checking approach described above, the formula is translated to CTL as $\Phi((\mathbf{X}p)\ \mathbf{U}\ q) = \mathbf{E}((\mathbf{EX}p)\ \mathbf{U}\ q)$.

The model fulfils $K(s_0) \models_{\mathrm{CTL}} \mathbf{E}((\mathbf{EX}p)\ \mathbf{U}\ q)$, because the path $\pi = s_0.s_1.s_2.s_3.s_4\ldots$ fulfils $(\mathbf{EX}p)\ \mathbf{U}\ q$. This is true because the states s_0, s_1, s_2, s_3 each fulfil $\mathbf{EX}p$, and in s_4, proposition q is fulfilled. Note that in state s_2, formula $\mathbf{EX}p$ holds because the outgoing path $s_2.s_5.s_6\ldots$ fulfils $\mathbf{X}p$. Path π, however, is not a witness for the LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$, since $s_2.s_3.s_4\cdots \not\models_{\mathrm{LTL}} \mathbf{X}p$. Also for path $\pi' = s_0.s_1.s_2.s_5.s_6\ldots$, the LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$ is not fulfilled, because s_6 neither fulfils p nor q.

Summarising, the CTL-based model checking approach yields a false alarm when trying to prove the absence of a witness for LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$. □

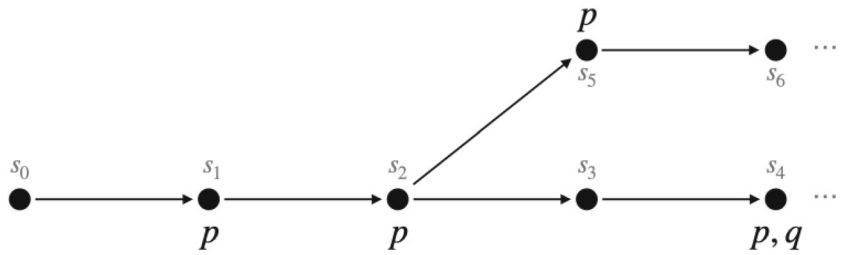


Fig. 11. Model fulfilling $\mathbf{E}((\mathbf{EX}p)\ \mathbf{U}\ q)$ but not $(\mathbf{X}p)\ \mathbf{U}\ q$.

```
    function checkLTL(in π : S*; in φ : LTL) : 𝔹
    begin
1     if φ = f is a positive first-order formula then
2       checkLTL := |[f]|_0;
3     elseif φ = ¬f is a negated first-order formula then
4       checkLTL := ¬|[f]|_0;
5     elseif φ = φ_0 ∧ φ_1 then
6       checkLTL := checkLTL(π, φ_0) ∧ checkLTL(π, φ_1);
7     elseif φ = φ_0 ∨ φ_1 then
8       checkLTL := checkLTL(π, φ_0) ∨ checkLTL(π, φ_1);
9     elseif φ = Xφ_0 then
10      checkLTL := (|π| > 1 ∧ checkLTL(π^1, φ_0));
11    elseif φ = φ_0 U φ_1 then
12      checkLTL := (checkLTL(π, φ_1) ∨ (|π| > 1 ∧ checkLTL(π, φ_0) ∧ checkLTL(π^1, φ_0 U φ_1)));
      endif
    end
```

Fig. 12. Algorithm for detecting false alarms.

To elaborate an algorithm for detecting false alarms, we recall from Section 4.2 that the sub-models to be checked in the context of data validation are rooted directed acyclic graphs (DAGs), with the entry point into the railway network as root. Each DAG corresponds to a subset of routes through the railway network, each route starting at the given entry point, but possibly ending in different exit points. The check whether a witness path π of a CTL formula is also a witness for the corresponding LTL formula φ can be performed using the finite LTL encodings presented above in Section 3.4.1, Tables 5 and 6.

Applying these rules, the algorithm shown in Fig. 12 can be used to check whether a path π is really a witness for a given LTL formula φ. The algorithm takes a finite path π and an LTL formula φ in negation normal form with temporal operators $\mathbf{X}$, $\mathbf{U}$ only as input. It returns `true` if and only if π is a model of φ in the interpretation $|\,[\varphi]\,|_0$ on π.[5]

Theorem 4.1 Algorithm *checkLTL*(π, φ) from Fig. 12 always terminates and returns `true` if and only if path π is an LTL witness for formula φ.

Proof. The proof is performed by structural induction over the formula syntax. To perform the inductive step, it is necessary to prove a slightly more general statement than that of the theorem. Let $k = |\ \pi\ | - 1$ be the last defined index of π. Then we prove that

$$\forall\, i \in \{0, \ldots, k\}.\ (\pi^i \models_{\mathrm{LTL}} g) \text{ if and only if } \mathit{checkLTL}(\pi^i, g) \tag{3}$$

holds for every sub-fromula g of φ, including φ itself. If (3) has been proven, the theorem follows by applying (3) with $i = 0$ and $g = \varphi$.

[5] We use Pascal-style notation: The return value is specified by means of an assignment to the function name.

Base case. Let g be an unquantified first-order expression and consider path segment π^i with $i \in \{0, \ldots, k\}$. For the evaluation result $checkLTL(\pi^i, g)$, there are two cases to distinguish. (a) If $g = f$ is positive, then the checking result is produced in line 2: The function returns `true` if and only if the formula evaluates to `true` in the path segment's first state $\pi(i)$. (b) If $\varphi = \neg f$ is negated, so that f is a positive formula, the checking result is produced in line 4: The function returns `true` if and only if the formula f evaluates to `false` in $\pi(i)$. Both cases conform to the LTL evaluation semantics specified for finite paths π in Table 5. The algorithm terminates immediately after having executed lines 2 and 4, respectively. This proves the base case for (3).

Induction hypothesis. Suppose that for sub-formulae φ_0 and φ_1 of φ, the calls to $checkLTL(\pi^i, \varphi_0)$ and to $checkLTL(\pi^i, \varphi_1)$ terminate for each $i \in \{0, \ldots, k\}$ and fulfil (3).

Induction step. We have to show that, under assumption of the induction hypothesis, termination is guaranteed and (3) holds for formulae $g = \varphi_0 \wedge \varphi_1$, $g = \varphi_0 \vee \varphi_1$, $g = \mathbf{X}\varphi_0$, and $g = \varphi_0 \mathbf{U} \varphi_1$.

Case $g = \varphi_0 \wedge \varphi_1$. In this case line 6 of the algorithm applies, and $checkLTL(\pi^i, \varphi_0 \wedge \varphi_1)$ returns `true` if and only if $checkLTL(\pi^i, \varphi_0) \wedge checkLTL(\pi^i, \varphi_1)$ holds, which means that both $checkLTL(\pi^i, \varphi_0)$ and $checkLTL(\pi^i, \varphi_1)$ evaluate to `true`. Now the induction hypothesis implies that this is the case if and only if $\pi^i \models_{\text{LTL}} \varphi_0$ and $\pi^i \models_{\text{LTL}} \varphi_1$. Applying the semantic rule for the $\wedge$-operator in Table 5, we conclude that this holds if and only if $\pi^i \models_{\text{LTL}} \varphi_0 \wedge \varphi_1$. This proves the $\wedge$-case for (3). Termination is ensured, since at most $checkLTL(\pi^i, \varphi_0)$ and $checkLTL(\pi^i, \varphi_1)$ are executed, and both calls terminate according to the induction hypothesis.

Case $g = \varphi_0 \vee \varphi_1$ is verified in analogy to the $\wedge$-case.

Case $g = \mathbf{X}\varphi_0$. For this case, line 10 of the algorithm applies. For the special case where path π^i has length 1, `false` is returned which conforms to the evaluation rule in Table 6 for the case $i = k$. If the path length is greater than 1, the call checkLTL(π^{i+1}, φ_0) is performed which, due to the induction hypothesis, returns `true` if and only if $\pi^{i+1} \models_{\text{LTL}} \varphi_0$. Summarising, $checkLTL(\pi^i, \mathbf{X}\varphi_0)$ returns `true` if and only if $i < k$ and $\pi^{i+1} \models_{\text{LTL}} \varphi_0$. Now the semantic rule for the $\mathbf{X}$-operator in Table 6 states that this is the case if and only if $\pi^i \models_{\text{LTL}} \mathbf{X}\varphi_0$. This proves the $\mathbf{X}$-case for (3). Termination is ensured, since, if $checkLTL(\pi^i, \mathbf{X}\varphi_0)$ is called at all in line 10, its termination is guaranteed by the induction hypothesis.

Case $g = \varphi_0 \mathbf{U} \varphi_1$. For the $\mathbf{U}$-case, line 12 applies. There, it can be seen that the algorithm exactly implements the recursive rule for the $\mathbf{U}$-operator specified in Table 6. However, we cannot apply the induction hypothesis to operand $checkLTL(\pi^1, \varphi_0 \mathbf{U} \varphi_1)$, since this references $\varphi_0 \mathbf{U} \varphi_1$. Therfore, we perform another induction over the length of the path segments π^i. For length 1 (i.e. $i = k$), line 12 returns $checkLTL(\pi^i, \varphi_1)$. Termination is ensured by the structural induction hypothesis. Moreover, the latter implies that this call returns `true` if and only if $\pi^i \models_{\text{LTL}} \varphi_1$, which conforms to the semantic rule for $\mathbf{U}$ in the case $i = k$. Now suppose that termination is ensured and (3) holds for $g = \varphi_0 \mathbf{U} \varphi_1$ on all path segments $\pi^k, \ldots, \pi^{k-i_0}$, $0 \leq i_0 < k$. We need to show that then that termination is guaranteed and (3) also holds for path segment π^{k-i_0-1}. From line 12 and the fact that $| \pi^{k-i_0-1} | = | \pi | - k + i_0 + 1 = i_0 + 2 > 1$ we conclude that the return value of $checkLTL(\pi^{k-i_0-1}, \varphi_0 \mathbf{U} \varphi_1)$ is $\big(checkLTL(\pi^{k-i_0-1}, \varphi_1) \vee (checkLTL(\pi^{k-i_0-1}, \varphi_0) \wedge checkLTL(\pi^{k-i_0}, \varphi_0 \mathbf{U} \varphi_1))\big)$. To the first two operands of this expression, we can apply the structural induction hypothesis, and to the third operand, the hypothesis of the induction over the length of the path segment. This implies that $checkLTL(\pi^{k-i_0-1}, \varphi_0 \mathbf{U} \varphi_1)$ terminates and returns `true` if and only if $\pi^{k-i_0-1} \models_{\text{LTL}} \varphi_1 \vee (\pi^{k-i_0-1} \models_{\text{LTL}} \varphi_0 \wedge \pi^{k-i_0} \models_{\text{LTL}} \varphi_0 \mathbf{U} \varphi_1)$. This conforms to the semantic rule for the $\mathbf{U}$-operator as specified in Table 6, proves (3) for the $\mathbf{U}$-case, and completes the structural induction. □

Example 4.8 Applying the detection algorithm for false alarms from Fig. 12 to the finite path $s_0.s_1.s_2.s_3.s_4$ which is a witness of $K(s_0) \models_{\text{CTL}} \Phi((\mathbf{X}p)\mathbf{U}q)$ in Example 4.7 results in the recursive call tree shown in Fig. 13. As expected, the algorithm returns `false` for the call checkLTL$(s_0.s_1.s_2.s_3.s_4, (\mathbf{X}p)\mathbf{U}q)$, so this CTL witness for $\Phi((\mathbf{X}p)\mathbf{U}q)$ is a false alarm. □

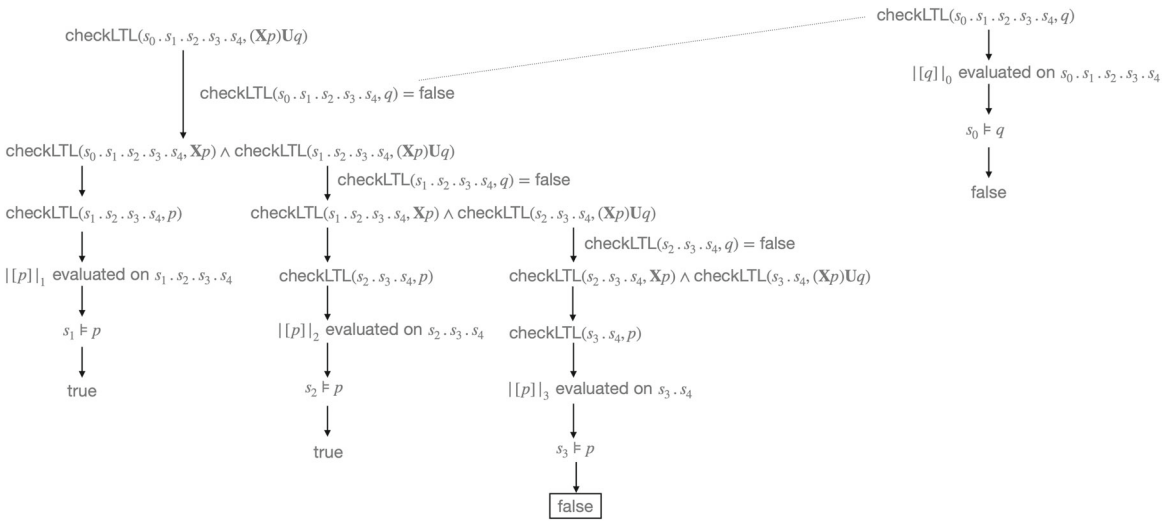


Fig. 13. Recursive call tree resulting from application of Algorithm from Fig. 12 to Example 4.7.

4.5. Parallelisation

The concept to use sub-models for verifying DVL-queries allows for parallelisation of checking activities. The concurrent checker design is shown in Fig. 14. At the user interface, a checking request is submitted to the DVL-Checker and received there by the request manager. The request consists of a set of IXL configuration models, and each model in the list is associated with a set of queries (LTL formulae specifying IXL configuration rule violations) to be checked against the model. Since IXL configurations are usually repeatedly accessed with different queries during a checking session, the sub-models created from each model are cached, so that sub-model creation is only required once per model and per session.

If a model in the checking request is referenced for the first time in the current session, or if a model has been updated, it is read by the sub-model generator from the model database, where all IXL configurations are stored in XML format. The generator creates a Kripke structure as described in Section 4.1 from the model which kept in memory until the sub-model creation has been completed. For each border element of the model, a job consisting of a reference to the Kripke structure and a border element identification is inserted into the job queue. Worker threads retrieve these jobs from the queue and execute the sub-model generation algorithm explained in Section 4.2. The resulting sub-models are cached.

Analogously, queries that are not yet contained in the query cache are parsed and transformed into CTL by worker threads exercising the query parser.

The checking requests for cached sub-models and associated queries are transferred by the request manager into the job-queue. Each job consists of one sub-model reference and one query. For these jobs, the worker threads invoke a (sequential) CTL model checker running the algorithms described in Section 3.7. Several worker threads may execute CTL checks concurrently, each with a different pair of sub-model and query. If a CTL check yields a CTL witness for a potential rule violation, the witness is passed on to the false alarm filter described in Section 4.4, where it is checked whether the witness is also an LTL witness representing a "real" IXL configuration rule violation. The false alarms are discarded by the filter. The valid rule violations are presented by the request manager to the users.

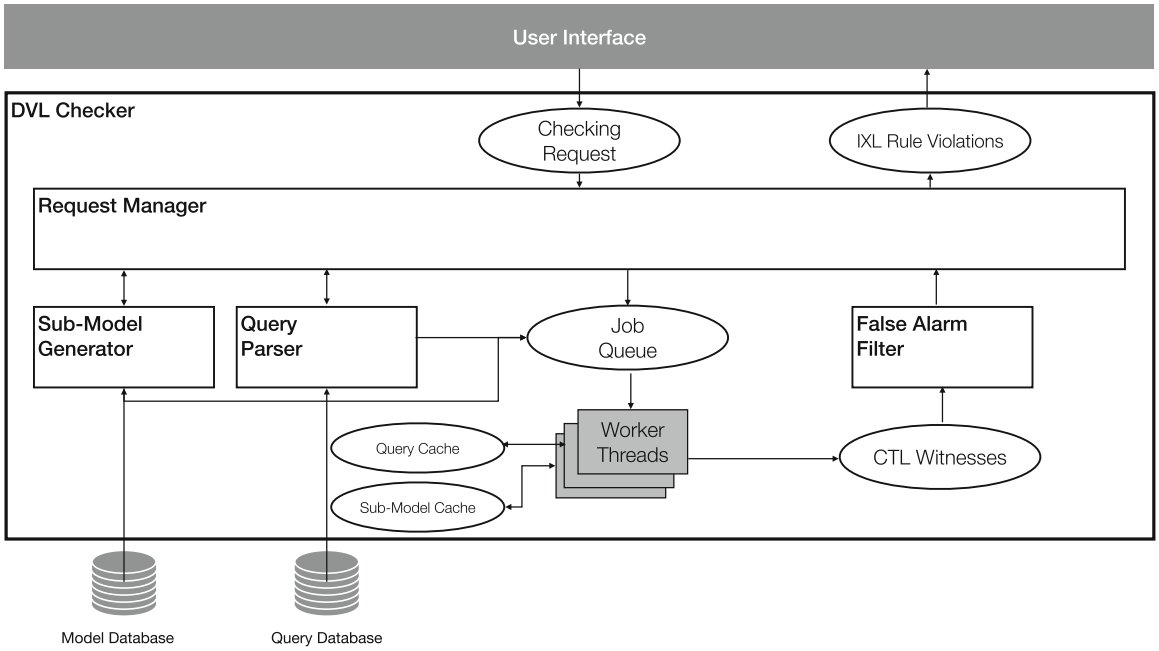


Fig. 14. DVL-Checker—architecture.

Table 8. Running time for query evaluation on sub-models for IXL test configurations provided by Siemens.

IXL Config.	# Elements	# Sub-Models	# Queries	t[ms]	t‖[ms]
TST02	333	8	18	149	65
TST03	73	2	18	25	14
TST04	176	2	18	182	94
TST05	313	19	18	577	240
TST06	412	11	18	1615	690
$\sum$				2548	1103

4.6. Evaluation

Running time for query evaluation

The efficiency of the CTL model checking algorithms in combination with the parallelisation allows for checking queries interactively, because the results can usually be obtained within a few seconds. For a more detailed evaluation, 5 IXL test configurations were provided by Siemens, see Table 8. The evaluation has been performed on a Linux PC with Intel(R) Core(TM) i7-6700HQ CPU (2.60 GHz, 4 cores), and 32GB main memory. The worker queues allowed for up to 7 concurrent threads for processing the 18 queries on each sub-model. In Table 8, the columns 2, 3, and 4 list the number of elements contained in each model, the number of sub-models, and the number of queries (i.e. CTL formulae $\Phi(f)$, where f is an LTL formula specifying a rule violation). In the fifth column marked by **t[ms]**, the evaluation for sequential (single core, single-threaded) processing of 18 queries on each sub-model is listed. In the last column marked by **t‖[ms]**, the running time for concurrent query evaluation is listed. With the available hardware, the concurrent evaluation more than doubles the speed. All 18 queries[6] could be evaluated on all 5 models less than 1.2 seconds. It should be noted that with today's cloud technologies, further speed-up could be achieved by running the evaluation on a cloud server with more CPU cores.

No false alarms have been encountered with the DVL queries checked so far on the IXL configurations provided by Siemens.

[6] Note that the full catalogue of IXL configuration rules consists of several hundred rules.

Table 9. Running time for sub-model creation from IXL test configurations.

IXL Config.	# Sub-Models	t[ms]	t‖[ms]
TST02	8	15	5
TST03	2	3	2
TST04	2	7	4
TST05	19	40	13
TST06	11	55	17
$\sum$		120	41

Running time for sub-model creation.

The sub-models are created separately and kept in main memory, whenever a new model is provided, or when an existing model has been updated. This allows for running queries against sub-models at different points in time without having to re-generate the sub-models for every query. Table 9 shows the time measurements performed during the sub-model generation for the IXL test configurations provided by Siemens. In the third column labelled by **t[ms]**, the running time for sequential sub-model creation is shown. In the fourth column, the running time for concurrent sub-model creation with 7 worker threads on the same hardware specified above is listed. For the larger models, the parallelisation approximately triples the speed of the sub-model creation. It can be seen from Table 9 that sub-model creation is not a critical timing factor for the DVL-Checker: the creation of all sub-models from all models took less than 0.5 seconds.

Comparison with bounded model checking approach.

The bounded model checking version used before as described in [HPP13] could also produce witnesses for faulty configurations in acceptable time (less than 10 seconds for models that are comparable to the ones shown in Table 8), but was unable to prove the *absence* of errors, due to running time that was exponential in the length of the search paths and very high memory consumption. Moreover, the approach investigated in [HPP13] operated on the complete Kripke model representing the whole network. This required the utilisation of *existentially quantified* LTL queries [MP92]: intuitively speaking, the existential quantification is needed when operating on the complete model to identify neighbouring track elements *in driving direction*. This increased the running time for query processing in a significant way.

5. Related work

Data validation for railway interlocking systems is a well-established V&V task in railway technology. At the same time, it is a very active research field, since the complexity of today's IXL configurations require a high degree of automation for checking their correctness. There seems to be an agreement among the research communities that hard-coded data validation programs are inefficient, due to the large number of rules to be checked and the frequent adaptations and extensions of rules that are necessary to take into account the requirements of different IXLs. These observations are confirmed by numerous publications on IXL data validation, such as [BDP12, HPP13, HSL16, FLFO17, KZC19].

It is interesting to point out that some V&V approaches for IXLs do not explicitly distinguish between data validation and the verification of dynamic IXL behaviour; this is the case, for example, in [CK16, KZC19]. We agree, however, with [FLFO17] (and have applied this principle, for example, in [HHP17, HØ16]), where it is emphasised that data validation should be a separate activity in the IXL V&V process. This assessment is motivated by the analogy to software verification, where the correctness of static semantics—this corresponds to the IXL configuration data—is verified before the correctness of dynamic program behaviour—this corresponds to the dynamic IXL behaviour—is analysed.

As observed in [BtBF+18], the B-Method and its variant Event-B are the most widely used formal methods in the railway domain. This holds for both industrial and academic applications. This success story started with

the application of B for the development of the driverless Paris Metro 14, where B was used for both software verification and data validation [BBFM99]. The core B formalism is based on quantified first-order logic and provides a theorem prover for automated and interactive verification of correctness properties. As reported in [LBL12], the original theorem prover was less well-suited for data validation, where constraints on—potentially very complex—data types need to be verified. Therefore, data validation approaches based on the B tool family usually rely on model checking; we name [LBL12, BDP12, HSL16, FLFO17, KZC19] as noteworthy examples for this fact. For model checking purposes in this context, the ProB tool seems to be the most widely used [HSL16].

The methodology and tool support described in the present paper differs significantly from the B approaches to data validation: while the latter require specifications in first-order logic, our approach is based on temporal logic. Moreover, our methodology is strictly specialised on geographic interlocking systems, while—in principle—the B-methods can be applied to any type of IXL technology. Our more restricted approach, however, comes with the advantage that rule specifications are simpler to construct than in B, since the temporal logic formulae do not require quantification over variables. Moreover, the sub-model construction technique used in our methodology ensures that the proper verification by CTL model checking is always fully automatic and fast. Since the ProB approach described in [HSL16] specialises on Thales/Alsthom railway control systems, while our approach is focused on geographical Siemens interlocking systems, the IXL configuration data to be validated, as well as the validation rules, differ significantly. Therefore, we cannot state whether one approach is superior to the other; it can only be said that both approaches work with sufficient effectiveness.

The utilisation of sub-models has also gained attention in the field of verification of route-based interlocking systems [JMN+14]. There, sub-models called *cones* are used to identify sub-networks from where the safety of a given set of track elements could potentially be violated at runtime. This construction, however, differs significantly from our sub-model construction: we always start at an entry point and unfold an acyclic graph in driving direction from the entry point to all reachable exit points, whereas the cones in [JMN+14] are constructed "backwards" from a set of track elements that are neither entry, nor exit points. The difference in the construction is motivated by the different verification objectives: our presentation aims at data validation and disregards dynamic safety aspects, because the latter can only be verified *after* the consistency of the IXL configuration data has been shown. In [JMN+14], however, behavioural safety of route-based interlocking systems is investigated, which is quite a different objective.

An general overview of trends in formal methods applications to railway signalling can be found in [Bjø03, FFM12, BtBF+18]. Many other research groups have been using model-checking for the behavioural verification of interlocking systems. In [FMGF11] a systematic study of applicability bounds of the symbolic model-checker NuSMV and the explicit model checker SPIN showed that these popular model checkers could only verify small railway yards. Several domain-specific techniques to push the applicability bounds for model checking interlocking systems have been suggested. Here we will just mention some of the most recent ones. In [Win12] Winter pushes the applicability bounds of symbolic model checking with NuSMV by optimising the ordering strategies for variables and transitions using domain knowledge about the track layout. Fantechi suggests in [Fan12] to exploit a distributed modelling approach to geographical interlocking systems and break the verification task into smaller tasks that can be distributed to multiple processors such that they can be verified in parallel. In [MNR+13], it is suggested to shrink the state space using abstraction techniques reducing the number of track sections and the number of trains. In [HHP17], we have shown that bounded model checking in combination with k-induction can cope with the size of real-world route-based interlocking systems for verifying their behaviour. As an alternative to the B-family, the RAISE tool offers the possibility to perform combined verification by theorem proving and model checking [GH18].

6. Conclusion

We have presented an efficient model checking approach and associated tool support for data validation of geographical interlocking systems. The tool is fast enough to uncover violations of configuration rules or prove the absence of rule violations interactively, while working on a configuration: all checking results for IXL configurations provided by Siemens Mobility were calculated within a few seconds.

The checking speed has been achieved by translating LTL formulae specifying rule violations to CTL formulae and using the "classical" global CTL model checking algorithms. It has been shown that for the class of LTL formulae specifying rule violations, CTL model checking is an over-approximation for the (slower) alternative to check for witnesses of LTL formulae directly. Therefore, the absence of CTL witnesses proves the absence of

path segments fulfilling the original rule violation formula specified in LTL. Since CTL is an over-approximation, solutions to the CTL formulae may turn out to be false alarms. Therefore, an algorithm has been presented to check CTL witnesses, whether they are also witnesses for the original LTL formulae specifying the rule violations. If this is not the case, the CTL witness is a false alarm and may be discarded.

Further speed-up has been achieved by running checks concurrently on configuration sub-models augmented by auxiliary attributes, instead of performing a single check on the full model.

The concepts and algorithms presented here have been implemented in the DVL-Checker tool which is used by Siemens for the validation of IXL configurations in new interlocking systems provided by Siemens for Belgian railways.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

[BBFM99] Behm P, Benoit P, Faivre A, Meynadier J-M (1999) Météor: A successful application of B in a large project. In: Wing JM, Woodcock J, Davies J (eds) FM'99—Formal methods, world congress on formal methods in the development of computing systems. Toulouse, France, September 20–24, 1999, Proceedings, Volume I, volume 1708 of lecture notes in computer science. Springer, pp 369–387

[BDP12] Badeau F, Doche-Petit M (2012) Formal data validation with event-B. arXiv:1210.7039 [cs], October

[BHJ+06] Biere A, Heljanko K, Junttila T, Latvala T, Schuppan V (2006) Linear encodings of bounded LTL model checking. Log Methods Comput Sci 2(5), November. arXiv:cs/0611029

[Bjø03] Bjørner D (2003) New results and current trends in formal techniques for the development of software for transportation systems. In: Proceedings of the symposium on formal methods for railway operation and control systems (FORMS'2003), Budapest/Hungary. L'Harmattan Hongrie, May 15–16

[BtBF+18] Basile D, ter Beek MH, Fantechi A, Gnesi SM, Piattino FA, Trentini D, Ferrari A (2018) On the industrial uptake of formal methods in the railway domain. In: Furia CA, Winter K (eds) Integrated formal methods, lecture notes in computer science. Springer International Publishing, pp 20–29

[CEN11] CENELEC (2011) EN 50128:2011 railway applications—communication, signalling and processing systems—software for railway control and protection systems

[CGP99] Clarke EM, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge

[CK16] Celebi BT, Kaymakci OT (December 2016) Verifying the accuracy of interlocking tables for railway signalling systems using abstract state machines. J Mod Transp 24(4):277–283

[CLRS09] Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. The MIT Press

[Fan12] Fantechi A (2012) Distributing the challenge of model checking interlocking control tables. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Applications and case studies, volume 7610 of lecture notes in computer science. Springer, Berlin, pp 276–289

[FFM12] Fantechi A, Fokkink W, Morzenti A (2012) Some trends in formal methods applications to railway signaling. In: Formal methods for industrial critical systems. Wiley, pp 61–84

[FLFO17] Fredj M, Leger S, Feliachi A, Ordioni J (2017) OVADO. In: Fantechi A, Lecomte T, Romanovsky A (eds) Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification, lecture notes in computer science. Springer International Publishing, pp 87–98

[FMGF11] Ferrari A, Magnani G, Grasso D, Fantechi A (2011) Model checking interlocking control tables. In: Schnieder E, Tarnai G (eds) Proceedings of formal methods for automation and safety in railway and automotive systems (FORMS/FORMAT 2010), Braunschweig, Germany. Springer

[GH18] Geisler S, Haxthausen AE (2018) Stepwise development and model checking of a distributed interlocking system—using RAISE. In: Havelund K, Peleska J, Roscoe B, de Vink EP (eds) Formal methods—22nd international symposium, FM 2018, held as part of the federated logic conference, FloC 2018, Oxford, UK, July 15–17, 2018, Proceedings, volume 10951 of lecture notes in computer science. Springer, pp 277–293

[HHP17] Hong LV, Haxthausen AE, Peleska J (2017) Formal modelling and verification of interlocking systems featuring sequential release. Sci Comput Program 133:91–115

[HØ16] Haxthausen AE, Østergaard PH (2016) On the use of static checking in the verification of interlocking systems. In: Leveraging applications of formal methods, verification and validation: discussion, dissemination, applications, Part II, volume 9953 of lecture notes in computer science. Springer International Publishing AG, pp 266–278

[HPP13] Haxthausen AE, Peleska J, Pinger R (2013) Applied bounded model checking for interlocking system designs. In: Counsell S, Núñez M (eds) SEFM workshops, volume 8368 of lecture notes in computer science. Springer, pp 205–220

[HSL16] Hansen D, Schneider D, Leuschel M (2016) Using B and ProB for data validation projects. In: Butler M, Schewe K-D, Mashkoor A, Biro M (eds) Abstract state machines, alloy, B, TLA, VDM, and Z, lecture notes in computer science. Springer International Publishing, pp 167–182

[JMN+14] James P, Moller F, Nga NH, Roggenbach M, Schneider SA, Treharne H (2014) Techniques for modelling and verifying railway interlockings. Int J Softw Tools Technol Transf 16(6):685–711

[KZC19] Keming W, Zheng W, Chuandong Z (2019) Formal modeling and data validation of general railway interlocking system. WIT Trans Built Environ 181

[LBL12] Lecomte T, Burdy L, Leuschel M (2012) Formally checking large data sets in the railways. CoRR, abs/1210.6815

[LMP10] Laroussinie F, Meyer A, Petonnet E (2010) Counting LTL. In: Markey N, Wijsen J (eds) TIME 2010—17th international symposium on temporal representation and reasoning, Paris, France, 6–8 September 2010. IEEE Computer Society, pp 51–58

[MNR+13] Moller F, Nguyen HN, Roggenbach M, Schneider S, Treharne H (2013) Defining and model checking abstractions of complex railway models using CSP‖B. In: Biere A, Nahir A, Vos T (eds) Hardware and software: verification and testing, volume 7857 of lecture notes in computer science. Springer, Berlin, pp 193–208

[MP92] Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems—specification. Springer

[Pac02] Pachl J (2002) Railway operation and control. VTD Rail Publishing, January

[Pel20] Peleska J (2020) New distribution paradigms for railway interlocking. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation: applications—9th international symposium on leveraging applications of formal methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III, volume 12478 of lecture notes in computer science. Springer, pp 434–448

[PKHP19] Peleska J, Krafczyk N, Haxthausen AE, Pinger R (2019) Efficient data validation for geographical interlocking systems. In: Dutilleul SC, Lecomte T, Romanovsky AB (eds) Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification—third international conference, RSSRail 2019, Lille, France, June 4–6, 2019, Proceedings, volume 11495 of lecture notes in computer science. Springer, pp 142–158

[Sis94] Sistla AP (1994) Safety, liveness and fairness in temporal logic. Form Aspects Comput 6(5):495–511

[Son18] Steffens S, Siemens Mobility GmbH (2018) Safety@COTS multicore, distributed smart safe system DS3. In: Innovationstag ETCS stellwerk smartrail 4.0. Presentation Slides, pp 35–47

[Win12] Winter K (2012) Symbolic model checking for interlocking systems. In: Railway safety, reliability and security: technologies and system engineering. IGI Global, pp 298–315

Received 30 August 2020

Accepted in revised form 17 May 2021 by Alessandro Fantechi and Jim Woodcock

Published online 10 August 2021