# Improving Online Railway Deadlock Detection using a Partial Order Reduction

Bjørnar Luteberget
SINTEF Digital AS, Oslo, Norway
bjornar.luteberget@sintef.no

Although railway dispatching on large national networks is gradually becoming more computerized, there are still major obstacles to retrofitting (semi-)autonomous control systems. In addition to requiring extensive and detailed digitalization of infrastructure models and information systems, exact optimization for railway dispatching is computationally hard. Heuristic algorithms and manual overrides are likely to be required for semi-autonomous railway operations for the foreseeable future.

In this context, being able to detect problems such as deadlocks can be a valuable part of a runtime verification system. If bound-for-deadlock situations are correctly recognized as early as possible, human operators will have more time to better plan for recovery operations. Deadlock detection may also be useful for verification in a feedback loop with a heuristic or semi-autonomous dispatching algorithm if the dispatching algorithm cannot itself guarantee a deadlock-free plan.

We describe a SAT-based planning algorithm for online detection of bound-for-deadlock situations. The algorithm exploits parallel updates of train positions and a partial order reduction technique to significantly reduce the number of state transitions (and correspondingly, the sizes of the formulas) in the SAT instances needed to prove whether a deadlock situation is bound to happen in the future. Implementation source code and benchmark instances are supplied, and a direct comparison against another recent study demonstrates significant performance gains.

## 1 Introduction

The discrete elements used in the control of railway traffic, along with the tightly controlled, predictable environment, makes railway operations a good match for discrete mathematical modeling formalisms, such as classical planning, temporal planning, propositional logic, mixed-integer programming, etc. There is a large amount of literature on railway operations such as routing and scheduling in both the field of mathematical optimization (see [6], demonstrating the diversity and sophistication of the field) and in the field of formal methods (see [1] and [9]), analyzing both for safety and for performance.

Systems for automating re-routing and re-scheduling can have significant impact on a railway system's overall delay recovery (and consequently also significant economical impact). However, they have also proven to become intractable at the network scales required for modeling complex railway systems. Routing and scheduling algorithms will have the side-effect of detecting deadlocks (because a bound-for-deadlock system will not have any possible valid schedules), though that is seldom their main purpose. On-line deadlock detection as a separate analysis may be useful in the following scenarios:

- In **manual train dispatching**, trains may enter a bound-to-deadlocked state after certain crucial decisions made by a dispatcher. Although this happens very infrequently on European railways dominated by relatively short passenger trains, it has been known to happen. It is a more pressing issue in the North American railway systems (see [19]), which are more dominated by freight trains that may be much longer that the usual European case. If deadlocks arise in the actual real-time position of the trains, it will be valuable for human operators to be notified of this as

M. Farrell & M. Luckcuck (Eds.):
Third Workshop on Formal Methods for Autonomous Systems (FMAS2021)
EPTCS ??, 2021, pp. 1–18, doi:10.4204/EPTCS.??.??

early as possible, because extraordinary operational modes will typically be required to resolve the deadlock by pushing or pulling trains backwards out of the deadlocked area.

- In **semi-automatic or automatic dispatching**, and **semi-autonomous or autonomous train driving**, it is still unclear whether autonomous systems will in practice solve (NP-hard) optimization problems exactly, or whether they will use heuristic approximations. If the latter is the case, then they cannot in general promise deadlock freedom (which is in itself also an NP-hard problem, see [15]), and detecting future deadlock states may be valuable, either for use in development, testing, and as a run-time verification tool, or for internal verification inside a heuristic optimization algorithm. I.e., if deadlocks arise after applying the schedules produced by an approximate scheduling algorithm, then this result can be fed back into the scheduling algorithm to help it suggest a new deadlock-free schedule. A similar idea was used in [18, 17], where a heuristic algorithm is repaired using model checking to fix all possible deadlock situations possibly created for a specific infrastructure. An approach based on Petri net models solves a similar problem [10], though neither of these have on-line performance constraints. Also, in [11], model checking is used to avoid deadlocks in autonomous last-mile transportation.

For both of these scenarios, the real-time situation in the physical railway system changes every few seconds or minutes, so for an analysis algorithm to be useful it should have an upper limit on typical running times somewhere between 10 seconds and 1 minute. The recent study in [21] highlights that there are few effective approaches for determining online (i.e., on real-time data) whether a running railway system is currently headed towards a deadlocked state. Although all online planning approaches for solving re-routing and re-scheduling trains (see [8] for a survey of these) implicitly solve the deadlock problem, these approaches usually result in large and sophisticated optimization problem formulations that can be hard to solve exactly. Whenever such re-routing and re-scheduling systems resort to heuristic algorithms (such as in [14]) they cannot actually give a definite answer to whether the system is bound for deadlock. The algorithm presented in [21] uses a classical planning model solved with a mixed-integer programming (MIP) solver, and simplifies other operational aspects to focus in on the deadlock detection problem. Their approach calculates a bound on the number of transitions required and, when instantiating the model with this number of steps, the infeasibility of the resulting MIP problem indicates that the system is in a bound-for-deadlock state. However, the approach is not fully satisfactory because the number of planning steps required in the worst case makes the problem intractable in practice.

Classical planning problems can be solved by exact mathematical solvers (such as MIP and SAT solvers) by encoding the state of the system and the transitions between consecutive states. SAT-based planning is closely related to bounded model checking (BMC), and indeed there is a large overlap in the techniques used for planning as satisfiability and model checking (see [2], [24] and [4, Ch. 18 and 19]). The number of planning steps used when solving classical planning problems greatly impacts the performance of the solver (see [20]), but there is no general way to know how many transitions will be required to solve a classical planning problem. In some special cases, one can compute reasonable problem-specific upper (and lower) bounds (called the *completeness threshold* in BMC) and use that to instantiate the formula used in the mathematical solver. Another approach is to build the formula incrementally, adding more steps as required until the desired solution appears (see [22], [7]).

Parallel plans, i.e. allowing multiple actions to take place in the same transition, is an important technique for reducing the number of steps required to find a plan (see [20]). However, when the railway system is bound for a deadlock, having reduced the minimum length of the plan does not necessarily help much, as it is the *maximum* plan length that determines how many transitions are required to show a deadlock. In this paper, we show how to reduce both the minimum and the maximum plan length in a SAT-based planning model of railway operations by enforcing a *maximal progress constraint* that not only allows short plans but also forces them to have minimum length by forcing equivalent partial orders to have the same representation (called process semantics in [20], [12]).

Figure 1 shows a railway with four two-track stations connected by single-track lines. In the situation shown in State 0, a parallel plan could potentially have been found after only 4 transitions if the two long trains were short enough to meet at one of the stations, although the plan would require up to 26 actions to be executed. To show that the system is deadlocked using classical planning without the maximal progress constraint would then also require 26 transitions. Adding the maximal progress constraint that forces all allocations to happen as early as possible makes the SAT formula unsatisfiable (thus proving a bound-for-deadlock situation) already after 4 transitions.

This paper contains the following contributions:

- a state transition system encoding into propositional logic, based on our previous work in [16] and adapted for online railway deadlock detection (Section 3.1), allowing parallel application of planning operators both for different trains and for contiguous sections of a single train's path.
- successive algorithmic improvements on the base propositional encoding: returning early when finding the shortest valid plan (Section 3.2), applying a global progress constraint and returning early also when there are no possible actions (Section 3.3), and applying a maximal progress constraint that enforces a partial order reduction, further reducing transitions (Section 3.4).
- a demonstration of the improved performance of our SAT-based solver compared against running time and transitions count values from the literature (specifically [21]) (Section 4).

  The source code and problem instances are available online.

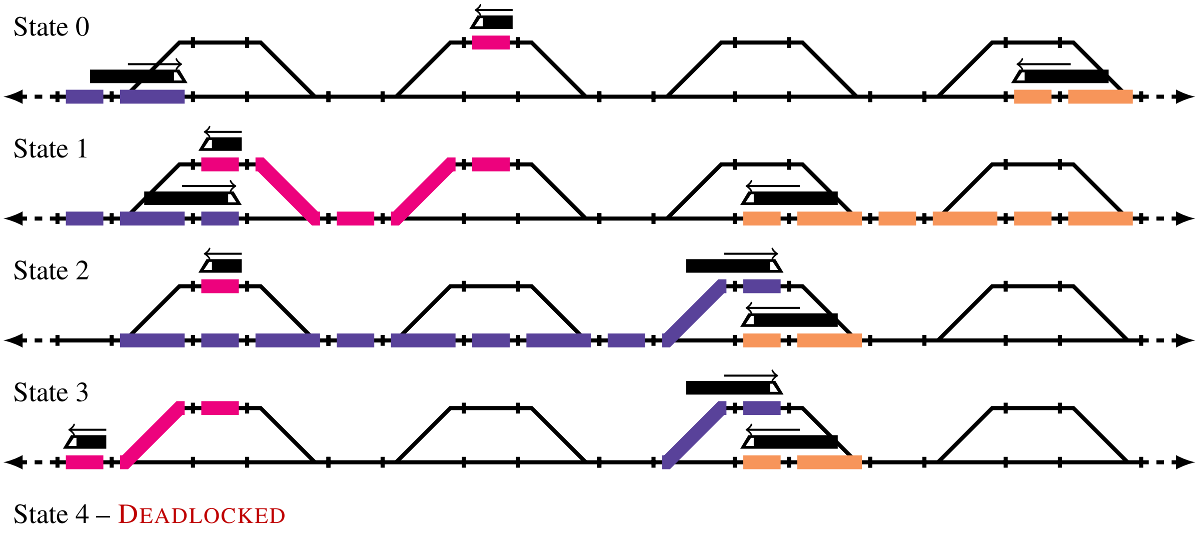


Figure 1: The railway infrastructure and trains shown in State 0 is bound for a deadlock, because the left-most and the right-most trains are too long to pass each other anywhere. States 1-3 are an example from the set of the longest possible plans for this system when applying the partial order reduction described in this paper. Note, for example, that the right-most train moves already in State 1 to the position where it will eventually block the left-most train. Using a pre-computed upper bound on the number of steps would instead require 26 transitions, making the resulting SAT instance harder to solve.

## 2 Problem Background

In this paper, we aim at solving the online railway deadlock detection problem, i.e. given a set of trains and their current locations, we want to determine whether the trains can reach their destinations or if the system is bound for a deadlock state.

The solution to this problem must show that exactly one of these two conditions hold:

- A plan for getting the trains to their destination exists, so there is no deadlock.
- There cannot exist *any* plan to get the trains to their destinations, so the trains will, by proceeding forward by any paths, eventually end up in a deadlocked state.

Note that, in contrast with deadlocks in computer programs, the goal of railway deadlock analysis is not to prove that there cannot be any deadlocks. For a computer program it is usually interesting to determine whether the program is completely safe from *any* deadlock occurring. In railway dispatching, on the other hand, there would usually be many opportunities for a human dispatching operator to cause a deadlock. The goal of railway deadlock analysis is to warn, as early as possible, when the operational situation changes to a state where there is no way to avoid reaching a deadlock in the future, i.e. the system is *bound for a deadlocked state*.

In the rest of this section, we describe the relevant domain background for railway operations and control systems needed to solve the deadlock problem, and then give a formal problem definition, and finally we give a brief introduction to SAT-based planning techniques which are used in Section 3 for solving the problem.

### 2.1 Railway Control Systems

A comprehensive model for railway operations analysis takes into account all of the following:

- Track and signalling component layout: railway tracks, switches (branching tracks), signal locations, detector locations, gradients, curves, tunnels, etc.
- The interlocking control system: takes commands to send a train from one location to another, and is responsible for allowing only movements which are safe from collision and derailment.
- Communication constraints: visual signals or radio communication are used to tell train drivers that the train may proceed.
- Train characteristics: the trains' lengths, acceleration and braking power, maximum speeds, etc. determine how they travel.

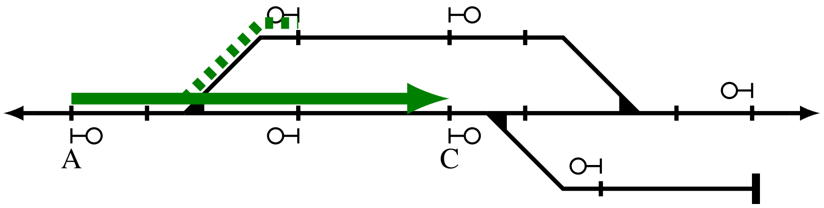


Figure 2: Elementary route AC from signal A to the adjacent signal C. The thick line indicates parts of the track on the train's path which are reserved for this movement, and the dashed lines indicate parts of the track outside the path which are also exclusively allocated.

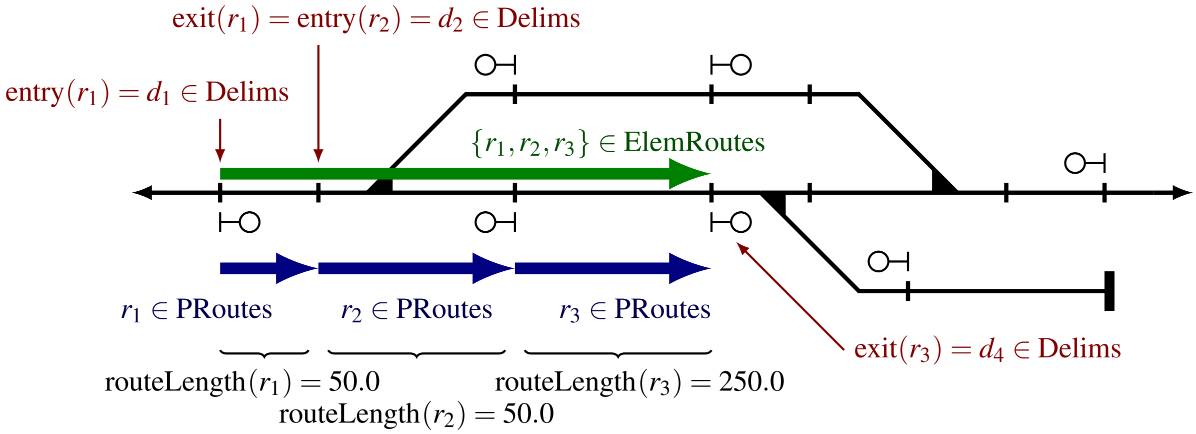

$\text{exit}(r_1) = \text{entry}(r_2) = d_2 \in \text{Delims}$
$\text{entry}(r_1) = d_1 \in \text{Delims}$
$\{r_1, r_2, r_3\} \in \text{ElemRoutes}$
$r_1 \in \text{PRoutes}$
$r_2 \in \text{PRoutes}$
$r_3 \in \text{PRoutes}$
$\text{exit}(r_3) = d_4 \in \text{Delims}$
$\text{routeLength}(r_1) = 50.0$
$\text{routeLength}(r_3) = 250.0$
$\text{routeLength}(r_2) = 50.0$

Figure 3: The abstracted infrastructure input used to represent the infrastructure in the planning SAT problem. The elementary route AC (see Figure 2) consists of three partial routes, $r_1, r_2, r_3$, connected through their shared delimiters ($\text{exit}(r_1) = \text{entry}(r_2)$, etc.). The exit signal of the elementary route is the delimiter $d_4$ (corresponding to signal C in Figure 2)

Fortunately, a much simpler model suffices to prove the absence or presence of a deadlock state. Since the interlocking only allows trains to move after exclusively allocating a discrete portion of the infrastructure, using the interlocking's discretization of the infrastructure into *elementary routes* (see Figure 2), is sufficient to solve the online deadlock detection problem. Note that elementary routes must be allocated as a whole, but can be partially released (see Figure 3). The parts are called *partial routes*. The trains' lengths are also relevant for deadlocks, since they determine whether an allocated route can be freed for use by other trains after the train has passed on to the next route.

Neither the communication constraints nor a train's dynamic characteristics are relevant for detecting a deadlock, since the existence of a valid plan does not depend on the time it takes for the trains to travel (so acceleration, braking, velocities, and signal sighting distance, are all irrelevant).

We disregard the use of safety zones (known as *overlaps* in British English, *durchrutschweg* in German) in this paper. Safety zones on real-world railways are unlikely to cause deadlocks. In many control systems they are simply freed after a timeout, and durations are irrelevant for the deadlock problem. Also, safety zone information does not figure in the benchmark instances we have used (from [21]). Handling of safety zones in a SAT-based planning model is described in [16].

## 2.2 Problem Definition

An instance of the online railway deadlock detection problem uses data about the infrastructure, and about the trains and their current locations. We define the route-based infrastructure model (summarized below as $I$) as the following data, similar to that used in in [16, Sec. 3.2]:

- A set of partial routes, PRoutes, and a set of elementary routes, $\text{ElemRoutes} \subset 2^{\text{PRoutes}}$, where each partial route belongs to exactly one elementary route.
- A set of route delimiters, Delims, each representing either a signal or a detector.
- Each partial route's length, $\text{routeLength} : \text{PRoutes} \to \mathbb{R}$.

- Each partial route's entry and exit delimiters, which take the null value at the model boundaries:

$$\text{entry}, \text{exit} : \text{PRoutes} \to \text{Delims} \cup \{\text{null}\}$$

- Conflicts between partial routes, $\text{Conflicts} \subset \text{PRoutes} \times \text{PRoutes}$.

In addition, we need the following train data (summarized below as $T$):

- A set of trains, Trains.
- The length of each train, $\text{trainLength} : \text{Trains} \to \mathbb{R}$.
- Each train's initial position, $\text{initialRoutes} : \text{Trains} \to 2^{\text{PRoutes}}$.
- Each train's final position alternatives, $\text{finalRoutes} : \text{Trains} \to 2^{\text{PRoutes}}$.

Note that a train that is required to leave the infrastructure at a specific model boundary specifies the route leading to that boundary as its final location, i.e. a route $r$ s.t. $\text{exit}(r) = \text{null}$, and the train can then leave the model (i.e., it does not need to stay in the route).

A plan $p \in \text{Plans}$ for moving the trains from their initial positions to their final positions is a sequence of pairs of trains and elementary routes,

$$\text{Plans} = (\text{Trains} \times \text{ElemRoutes})^*,$$

where the $*$ symbol denotes a sequence of elements. A plan is *correct* if it extends each train's initial routes with elementary routes which are correctly linked by consecutive delimiters (entry, exit) to lead them to one of their final position alternatives, and the plan can be executed without becoming *blocked*. A plan execution is blocked if, after allowing all trains to reach the end of their currently allocated paths, the route in the next step of the plan cannot be allocated because there is a conflict. Constraints ensuring correctness are described in Section 3.1 below.

**Definition 1.** The online railway deadlock detection problem $D = (I, T)$ is solved by a deadlock detection algorithm $d : D \to \{\textsc{Live}, \textsc{Dead}\}$, which returns LIVE if a *correct plan* exists, and DEAD otherwise.

### 2.3 SAT-based Planning and Parallel Plans

A classical planning problem is defined by a set of state variables $v \in V$, a set of actions $a \in A$, an initial state $\text{init} : V \to \mathbb{B}$ and a goal state defined on some variables $V_{\text{goal}} \subset V$, $\text{goal} : G \to \mathbb{B}$. For simplicity, we let all variable domains be Boolean ($v \in \mathbb{B}$). Actions have pre-conditions and post-conditions describing, respectively, the requirements for an action to take place and the effects of an action as a set of values assigned to state variables $V^a_{\text{pre}}, V^a_{\text{post}} \subset V$, $\text{pre} : V^a_{\text{pre}} \to \mathbb{B}$ and $\text{post} : V^a_{\text{post}} \to \mathbb{B}$. A solution to the planning problem is a sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ which transforms the initial state into the goal state.

A straight-forward encoding of classical planning into SAT creates a copy of the state space for each planning step, and allows application of exactly one action between each pair of consecutive states. In a single transition, variables that are not modified by the action can not change change, ensuring that the action's pre- and post-conditions are correctly satisfied. However, this encoding produces a large number of transitions when solving problem sizes encountered in real-life applications. If we instead allow multiple actions to be applied in each step, then we can reduce the number of steps required and thereby improve solver performance.

The actions in our railway representation are $A = \{a_{t,r} \mid t \in \text{Trains}, r \in \text{PRoutes}\}$, meaning that train $t$ allocates route $r$, where the preconditions are train consistency constraints and route exclusions, and post-conditions are occupations $o_r = t$. These conditions are formalized in the Section 3.1 below.

In fact, the total order of actions over-specifies a plan because many train movements happen concurrently and only a few of them have resource conflicts that makes their ordering meaningfully different. Starting from a plan $\pi = \langle a_{t_x,r_y}, \ldots \rangle$, we define a strict partial order $\prec_\pi$ as the smallest partial order containing:

1. each train's path, i.e. if train $t$ takes routes $r_1, r_2, r_3, \ldots\ldots$, then $a_{t,r_1} \prec_\pi a_{t,r_2},\ \ a_{t,r_2} \prec_\pi a_{t,r_3}, \ldots$.
2. for each pair of actions where $a_{t_1,r_1}$ precedes $a_{t_2,r_2}$ in the plan, if $(r_1, r_2) \in$ Conflicts, then $a_{t_1,r_1} \prec_\pi a_{t_2,r_2}$.

Two different plans $\pi_1 \neq \pi_2$ producing the same partial order $\prec_{\pi_1} = \prec_{\pi_2}$ are equivalent for the purposes of deadlock detection, a fact that we will exploit to reduce state space of the planning problem.

Following [20], we consider three approaches to find parallelizable actions, each corresponding (coincidentally) to one type of parallelization that we have used in the railway SAT encoding in Section 3.1:

- **$\forall$-steps**: When a set of actions can be applied in any order and still produce the same result, we may allow this set of actions to take place in the same transition.

  The most easily discovered parallel action opportunity we find in the route-based railway model is that actions that apply to different trains can always be applied in parallel. The mutual exclusion conditions on the route occupation variables are enough to make actions for different trains completely independently applicable, so this is a $\forall$-steps set of actions.

- **$\exists$-steps**: When a set of actions applied between two states is guaranteed to have at least one total order that transforms the first state into the second state, we may allow this set of actions to take place in the same transition. This is also called the *post-serializability* condition, and is potentially a very general framework for parallelizing planning actions.

  In the route-based railway model we exploit the following opportunity for $\exists$-steps parallelism: considering each train in the separately, if we have a train $t$ that travels on the sequence of routes $r_1, r_2, r_3$, the corresponding actions $a_{t,r_1}, a_{t,r_2}, a_{t,r_3}$ must apply in the same sequence. However, there is no problem in applying all these actions in the same step without representing the choice of order. Given a set of connected routes in an acyclic infrastructure, there is exactly one possible ordering of the actions, namely the one given by the directed route graph. So a set of routes forming a train path can be applied in parallel and *post-serialized* into a set of actions. Trains spanning long paths is also used in [13] for model checking, though there to abstract away train lengths.

- **Process semantics reduction**: Consider a classical planning problem with four actions $\alpha_1$, $\alpha_2$, $\alpha_3$, $\alpha_4$, where $\alpha_2, \alpha_3, \alpha_4$ must be executed in order, while $\alpha_1$ only needs to precede $\alpha_4$. Solving this problem with SAT-based planning requires only three steps (not four), since we have parallel action application of $\alpha_1$ with either $\alpha_2$ or $\alpha_3$. However, $\alpha_1$ can be still be applied either in the first step or the second step, and both describe the same partial order of actions (see Figure 4).

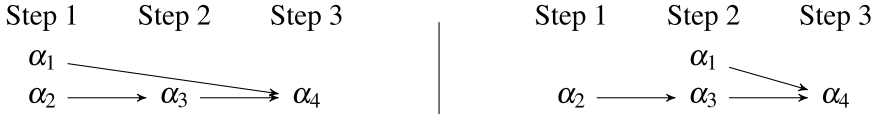


Figure 4: Two different state transition system plans for the actions $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. The arrows show the dependencies between actions. Whether $\alpha_1$ is executed in step 1 or 2 results in the same partial order.

We can remove these different but equivalent representations from the SAT problem by simply constraining actions to happen as early as possible. The encoding is described in Section 3.4 below. This property is called process semantics in [20], [12], and can have a significant impact on planning solver performance. Adding such a constraint, which we here call the *maximal progress constraint*, removes redundant plan representations that reduce to the same partial order $\prec$.

This constraint does not add parallelism between actions to reduce the *minimum* number of steps required to reach a state, but instead reduces the *maximum* number of steps required to execute a plan. For unsatisfiable planning problems (such as in a bound-for-deadlock state) it is a particular advantage to avoid a high maximum number of steps with little progress in each step.

We used this maximal progress constraint in previous work ([16]), where we enumerated all plans, although there we still used a heuristic for choosing the upper bound on the number of transitions.

# 3 SAT Encoding and Algorithms

In this section, we first present a SAT encoding of route-based railway as a state transition system (Section 3.1). Then we develop an algorithm for the online deadlock detection problem, using this SAT encoding, presented as three successively improving algorithms. First, a simple SAT-based planning algorithm with a statically computed upper bound on steps (Section 3.2), then we add a modification to detect deadlocks before reaching the static upper bound on steps (Section 3.3), and finally we add the maximal progress constraint that further reduces the number of steps until deadlock (Section 3.4).

## 3.1 State transition encoding

We now encode a state transition representation of the deadlock detection problem as propositional logic (a simplified and adapted version of the encoding in [16], ignoring safety zones). We write the propositional logic formula for a $k$-step unrolling of the transition relation as $\Phi_k = \bigwedge_{i=0}^{k} \phi_i$, where $\phi_0$ is the known initial state. We use well-known SAT encoding techinques for at-most-one constraints and one-hot encoded finite sets (see e.g. [23], [4, Ch. 2] or [5] for common encoding techniques and their tradeoffs). The formula $\phi_i$ for each state $i \geq 1$ of the system uses the following state variables:

1. Each partial route $r$ in step $i$ has an **occupancy status** $o_r^i$ which is either free ($o_r^i = \text{Free}$) or occupied by a specific train $t$ ($o_r^i = t$). We use a one-hot encoding to represent the finite set as Booleans so each $o_r^i = t$ is a variable in the SAT instance.
2. Each train $t$ in step $i$ has a Boolean representing **finished status** $f_t^i$, signifying that the train reaches its destination in state $i$ or before.

Note here that we do not need to add additional variables for each action $a_{t,r}$ in step $i$ (which we write $a_{t,r}^i$) when encoding this problem to conjunctive normal form: for any clause $C$, we can make the clause conditional on $a_{t,r}^i$ by writing $\big((o_r^{i-1}) \neq t \wedge (o_r^i = t)\big) \Rightarrow C$ as the equivalent $(o_r^{i-1} = t) \vee (o_r^i \neq t) \vee C$, which is a clause.

The following constraints apply to states and pairs of consecutive states to encode a *correct* plan:

- **Each train takes a single, contiguous path.**

  First, ensure that only one route from a given start delimiter may be taken at any time:

$$C_1^i = \bigwedge_{t \in \text{Trains}} \bigwedge_{d \in \text{Delims}} \text{atMostOne}\big(\{o_r^i = t \mid \text{entry(r)} = d\}\big)$$

Routes can only be allocated to a train when they extend some other route which was already allocated to the same train, i.e., consecutive routes must match so that the exit delimiter of one is the entry delimiter of the next:

$$C_2^i = \bigwedge_{t \in \text{Trains}} \bigwedge_{a \in \text{PRoutes}} (a_{t,a}^i \Rightarrow \bigvee_{\substack{b \in \text{PRoutes} \\ \text{entry}(a) = \text{exit}(b)}} o_b^i = t)$$

Note that this constraint ensures that the trains' allocation to routes *locally* forms a path in the graph of routes. In an acyclic infrastructure, this is sufficient. An approach for handling cyclic infrastructure is described in [16].

- **Conflicting routes are not active simultaneously.**

$$C_3^i = \bigwedge_{(a,b) \in \text{Conflicts}} \big((o_a^i = \text{Free}) \vee (o_b^i = \text{Free})\big)$$

- **Elementary routes are allocated as a unit.**

  The partial release feature of the interlocking system is handled by splitting each elementary route into separate routes for each component which can be released separately. The set ElemRoutes contains such sets of routes. Partial routes forming an elementary route must be allocated together:

$$C_4^i = \bigwedge_{t \in \text{Trains}} \bigwedge_{e \in \text{ElemRoutes}} \bigwedge_{r \in e} \left( a_{t,r}^i \Rightarrow \bigwedge_{r \in e} (o_r^i = t) \right)$$

- **Partial routes are only deactivated after a train has fully passed over them.**

  Routes are freed when sufficient length has been allocated ahead to fully contain the train.

$$C_5^i = \bigwedge_{t \in \text{Trains}} \bigwedge_{r \in \text{PRoutes}} (o_r^i = t) \Rightarrow \big((o_r^{i+1} \neq t) \Leftrightarrow \text{freeable}_t^i(r, \text{trainLength}(t))\big)$$

Note that the bidirectional implication sign on the right hand side means that deallocation is both allowed and required. The leftward part of this implication simplifies the train path consistency constraints because we do not need constraints to prevent trains from deallocating a section in the middle of their path and allocating an alternative path instead. The freeable formulas are produced by the following recursive function:

$$\text{freeable}_t^i(a, l) = \textbf{if } \text{exit}(a) = \text{null} \textbf{ or } l \leq 0 \textbf{ then } \top \textbf{ else}$$

$$\bigvee_{\substack{b \in \text{PRoutes} \\ \text{exit}(a) = \text{entry}(b)}} (o_b^i = t) \wedge \text{freeable}_t^i(b, l - \text{routeLength}(a)).$$

The function returns a disjunction of possible routes that can be taken after route $a$, i.e. from delimiter exit$(a)$, but if those routes are shorter than the length of the train, then they must be conjuncted with the freeable formula for the following route $b$. Note that the freeable function itself is not part of the SAT problem, but is used to compute a formula based on the static infrastructure data. No numerical representations are used in the SAT formulas.

Additionally, setting the finished status requires visiting one of the final routes (and $f_t^0 = \bot$):

$$C_6^i = \bigwedge_{t \in \text{Trains}} \left( (\neg f_t^{i-1} \wedge f_t^i) \Rightarrow \bigvee_{r \in \text{finalRoutes}(t)} o_r^i = t \right)$$

Combining these constraints, we get the state transition representation of state $i$ as $\phi_i = C_1^i \wedge C_2^i \wedge C_3^i \wedge C_4^i \wedge C_5^i \wedge C_6^i$. Finally, reaching the goal state at state $i$ (or before) is encoded as $G_i = \bigwedge_{t \in \text{Trains}} f_t^i$.

### 3.2 Algorithm 1: Upper-bounding $k$

Completeness thresholds (upper-bounding the number of transitions) are often too large for practical use, but for the online railway deadlock detection we require a complete algorithm, so we compute a completeness threshold here as a starting point for further algorithms. Let $n_j^k$ be the number of elementary routes on the longest path (recall that the infrastructure is assumed to be acyclic) from the starting position of train $t_j$ to its $k$th alternative destination in a specific problem instance $D = (I, T)$. Let $U = \sum_j \max_k n_j^k$.

**Proposition 1.** If any correct plan for $D$ exists, $\Phi_U \wedge G_U$ must be satisfiable.

This is indeed the approach taken in [21], where a lower and an upper bound on the number of planning steps is computed, by computing the lower and upper bound on the number of steps a single train in isolation would need to proceed to one of its destinations. However, always instantiating the system with $U$ planning steps is computationally expensive, and indeed [21] explores a varying number of transitions to see if a plan can be found with fewer transitions. Algorithm 1 implements a similar strategy where all $i = 1, 2, \ldots$ are tried successively, adding $\phi_i$'s to the SAT solver incrementally. The $G_i$'s are added temporarily using the *assumptions* interface that many incremental SAT solvers support.

```
Algorithm 1: Online railway deadlock detection using incremental $k$-bounded model checking
Input  : A problem instance $D = (I, T)$ and a bound $k$.
Output: DEAD if the system is bound for deadlock, LIVE otherwise.
1 let $i = 1$.
2 if $\Phi_i \wedge G_i$ is SAT, return LIVE
3 if $i < k$, increment $i$ and go to 2, else return DEAD
```

### 3.3 Algorithm 2: Early Return on Deadlocks

Instead of detecting deadlocks implicitly by the unsatisfiability of the formula when the number of transitions has reached the upper bound, we can decrease the number of required transitions by adding an explicit global progress constraint that requires that each step of the plan makes some amount of progress. In any planning step, even if it is the first one, if the progress constraint cannot be satisfied, then the system must be bound for a deadlock state.

At least one route must be allocated in each transition:

$$z_i = \bigvee_{t \in \text{Trains}} \bigvee_{r \in \text{PRoutes}} a_{t,r}^i, \qquad Z_i = \bigwedge_{j=1}^{i} z_j$$

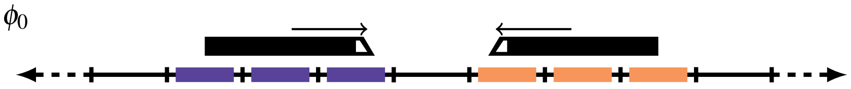
**Proposition 2.** If $\Phi_i \wedge Z_i$ is unsatisfiable for any $i$, then the instance is DEAD.

Algorithm 2 improves on Algorithm 1 by not needing to solve all $\Phi_i$ for $i$ all the way up to the explicitly computed completeness threshold $U$, but can instead terminate at the lowest number of steps where we can be sure that the system is in fact bound to be deadlocked. This is similar to the inductive proof algorithms from [22], but using incremental SAT like the "zig-zag" algorithm from [7].

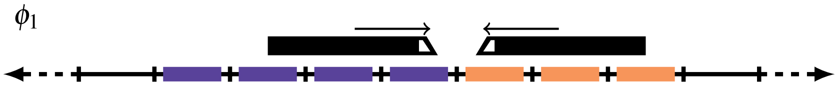
**Algorithm 2:** Online railway deadlock detection with global progress constraint

```
  Input  : A problem instance $D = (I, T)$.
  Output: DEAD if the system is bound for deadlock, LIVE otherwise.
1 let $i = 1$.
2 if $\Phi_i \wedge Z_i$ is UNSAT, return DEAD
3 if $\Phi_i \wedge Z_i \wedge G_i$ is SAT, return LIVE
4 increment $i$ and go to 2.
```

**Example 1.** Consider the following situation, where both trains are trying to exit at the model boundary in their traveling direction.



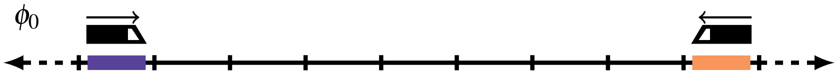
The situation is obviously bound for deadlock. However, because each train can take up to 5 steps to reach its model boundary, Algorithm 1 would require solving the transition system using $U = 10$ steps (corresponding to the sum of the length of the paths for each train) before concluding that the situation is DEAD. Algorithm 2 would return DEAD already after adding 2 transitions. The first transition would require either train to move ahead into the unoccupied route between them, and the second transition would then have no possible actions, so the formula $\Phi_2 \wedge Z_2$ is unsatisfiable.



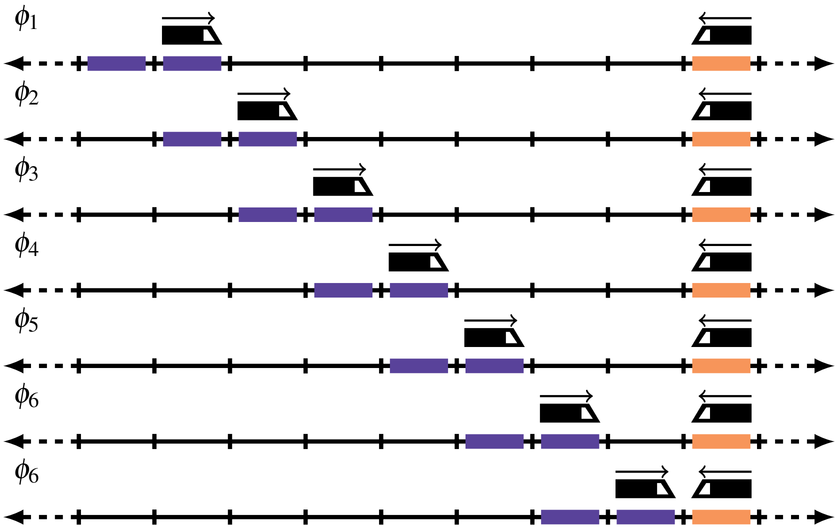
$\phi_2 \quad \rightarrow \quad$ DEAD

This is only the best case, though, there are still simple cases where the number of steps required to prove a deadlock still scales proportionally to the $k$ value in Algorithm 1.

**Example 2.** Consider the following situation (similar to Example 1):



This situation is obviously deadlocked in the same way as Example 1. However, here, Algorithm 2 requires 7 transition steps to terminate.



$\phi_7 \quad \rightarrow \quad$ DEAD

Note here that if the two trains were on separate tracks (so that they were not bound for deadlock), then the instance would have been proved LIVE using only one transition. The parallel actions that allow arriving at the LIVE result in fewer transitions does not correspondingly allow arriving at the DEAD result in similarly few transitions, as this example shows, because the *minimum* amount of progress in each transition is small.

### 3.4 Algorithm 3: Collapsing Equivalent Partial Orders

Looking in more detail at the actions taken by each train at each time step, we see that the global progress constraint $Z_i$ can be satisfied by trains moving forward by only one route per transition, as Example 2 demonstrates. We can force more progress to happen in each step using the following insight: for any $a_{t,r}$ taking place in state $i$ (except for $i = 1$), a route conflicting with $r$ must have been occupied in the previous state. If there was no such conflict in the previous state, performing $a_{t,r}$ in state $i-1$ would be possible, and would produce the same partial order $\prec$.

More precisely (and taking into account that the distinction between elementary routes and partial routes), we define the maximal progress constraint $P_i = \bigwedge_{j=2}^{i} p_j$, where:

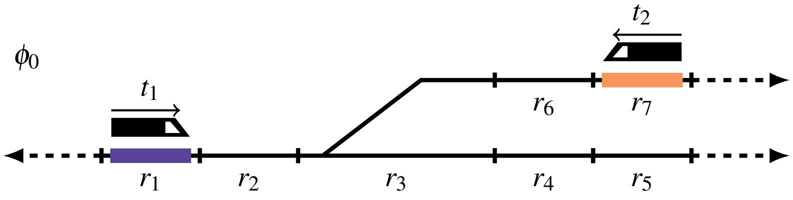
$$p_i = \bigwedge_{t \in \text{Trains}} \bigwedge_{\substack{e \in \text{ElemRoutes} \\ r \in e}} \left( \left( a^i_{t,r} \wedge \bigvee_{\substack{x \in \text{PRoutes} \\ \text{entry}(r) = \text{exit}(x)}} (o_x^{i-1} = t) \right) \Rightarrow \bigvee_{\substack{y \in e \\ z \in \text{PRoutes} \\ (y,z) \in \text{Conflicts}}} (o_z^{i-1} \neq t \wedge o_z^{i-1} \neq \text{Free}) \right)$$

$p_i$ encodes that if route $r$ is allocated to train $t$ in step $i$, and any of the preceding routes $x$ were allocated in the previous step, then at least one route conflicting with $r$ must have been allocated to another train in the previous step. $P_i$ is the conjunction of $p_i$'s from the initial state and up to state $i$. Note that $p_i$ does not apply for $i = 1$, because the initial state has known constant values for all variables, and the trains are not necessarily waiting for a route to become unallocated in the initial state.

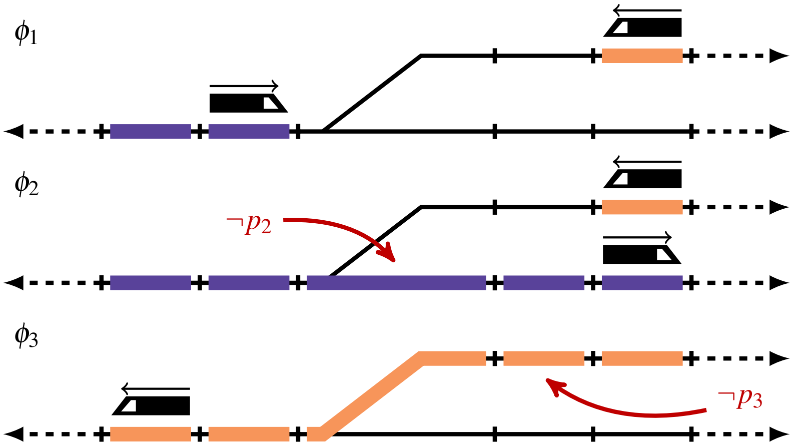
**Proposition 3.** The formula $\Phi_i \wedge Z_i \wedge P_i \wedge G_i$ admits all the same plans as $\Phi_i \wedge Z_i \wedge G_i$, modulo the plan partial order, i.e. adding the maximal progress constraint does not eliminate any valid plans.

To see why this is the case, consider a plan where, in step $i$, $p_i$ does not hold for a route $r$ allocated to a train $t$. This means that no route conflicting with $r$ was allocated (to another train) in state $i-1$. To fulfill $p_i$, we can simply allocate $r$ to $t$ in state $i-1$ instead of state $i$, which results in the same plan modulo the plan partial order. By repairing all solutions to $\Phi_k$ in this way, we see that adding the constraint $P_k$ preserves all plans modulo the partial order. See [20, Sec. 2.2.] for a more general and thorough treatment of this property.

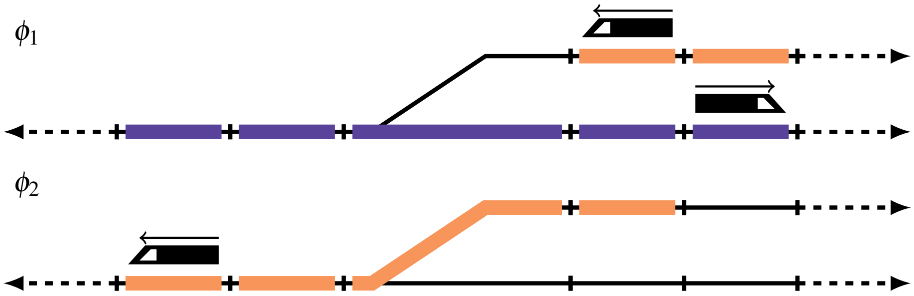
**Example 3.** Consider the following scenario:



Note that here we use the same route names for both travel directions, for simplicity. The following plan would satisfy the formula $\Phi_i \wedge Z_i$ used in Algorithm 2 with $i = 3$:



There are two violations of $p_i$ here, indicated by the red arrows. The first one in state 2, where the rightwards-headed train allocates the branching track when it was already free in $\phi_1$. The second one in state 3, where the leftwards-headed train allocates the route between its initial position and the branching route, when it was already free in $\phi_1$. An equivalent plan fulfilling $P_k$ uses only two planning steps:



Both of these solutions produce the same partial order $\prec$. We can visualize the partial order by drawing each member $(\alpha,\beta)$ of the transitive reduction of the partial order relation using lines with arrows pointing from the first component $\alpha$ to the second component $\beta$. Then both of the plans in the example above produce the following partial order:

$$\begin{tikzcd}
a_{t_1,r_1} \arrow[r] & a_{t_1,r_2} \arrow[r] & a_{t_1,r_3} \arrow[r] \arrow[dr] & a_{t_1,r_4} \arrow[r] & a_{t_1,r_5} & \\
& a_{t_2,r_7} \arrow[r] & a_{t_2,r_6} \arrow[r] & a_{t_2,r_3} \arrow[r] & a_{t_2,r_2} \arrow[r] & a_{t_2,r_1}
\end{tikzcd}$$

In general, adding the constraint $P_k$ forces a particular plan partial order to be represented in a unique way in the state space. Such constraints are known in the BMC literature as *partial order reductions*. Importantly, this constraint reduces the *maximum* number of steps that a set of trains can use to complete the possible set of plans, allowing deadlocks to become evident using a lower number of planning steps.

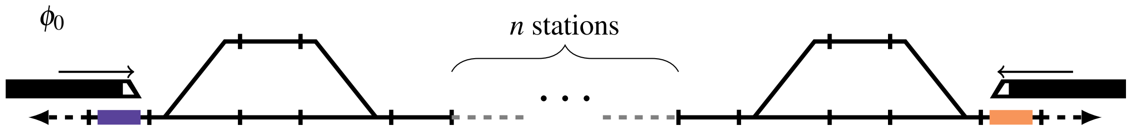
**Proposition 4.** If $\Phi_i \land Z_i \land P_i$ is unsatisfiable for any $i$, then the system is DEAD.

Algorithm 3 is defined in the same way as Algorithm 2, only with $\Phi_i$ replaced with $\Phi_i \land P_i$.

**Algorithm 3:** Online railway deadlock detection with partial order reduction

```
  Input   : A problem instance $D = (I,T)$.
  Output: DEAD if the system is bound for deadlock, LIVE otherwise.
1 let $i = 1$.
2 if $\Phi_i \land Z_i \land P_i$ is UNSAT, return DEAD
3 if $\Phi_i \land Z_i \land P_i \land G_i$ is SAT, return LIVE
4 increment $i$ and go to 2.
```

To demonstrate how $P_i$ changes the scaling properties of the deadlock detection algorithm in the best case, we have performed a scaling test using two trains starting on opposite sides of an infrastructure consisting of $n$ two-track stations, where none of the station routes are long enough for the trains to cross:



Running time results and number of steps for Algorithms 1, 2, and 3, for instances with increasing $n$ are shown in Table 1. We see that, in fact, for any number of stations and possible paths, 3 planning steps are sufficient to prove a deadlock involving two trains.

## 4 Performance Evaluation

We have used the problem instances from [21], and have evaluated algorithms 1, 2, and 3 described in this paper, comparing them to the reported worst-case number of transitions (called "*ticks*" in [21]) and running time results in [21]. The results are shown in Table 2. The running times for Algorithms 1, 2, 3 were measured on a laptop running Linux on an Intel i7-5500U CPU using the CaDiCaL SAT solver [3].

| Instance | | Algorithm 1 | | Algorithm 2 | | Algorithm 3 | |
|---|---|---|---|---|---|---|---|
| Stations | Routes | Steps | Time (s) | Steps | Time (s) | Steps | Time (s) |
| 2 | 16 | 10 (UB) | 0.00 | 7 | 0.00 | 3 | 0.00 |
| 4 | 32 | 22 (UB) | 0.15 | 13 | 0.19 | 3 | 0.00 |
| 6 | 48 | 34 (UB) | 2.80 | 19 | 1.70 | 3 | 0.00 |
| 8 | 64 | 46 (UB) | 18.90 | 25 | 6.70 | 3 | 0.00 |
| 10 | 80 | - | >60.00 | 31 | 35.90 | 3 | 0.00 |
| 20 | 160 | - | >60.00 | - | >60.00 | 3 | 0.01 |
| 50 | 400 | - | >60.00 | - | >60.00 | 3 | 0.04 |
| 100 | 800 | - | >60.00 | - | >60.00 | 3 | 0.28 |

Table 1: Scaling test with two trains headed in opposite directions on a sequence of two-track stations. Each station track is too short to contain any train, so the situation is bound for deadlock. Each row describes a problem instance with a number of two-track stations and corresponding number of routes in the problem. For each of the algorithms 1, 2, 3, two columns show the number of planning steps added to the problem before a conclusion is reached, and the running time of the algorithm. A timeout of 60 seconds was used.

| Instance | | | | Ticks MIP alg. | | Algorithm 1 | | Algorithm 2 | | Algorithm 3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | (reported in [21]) | | | | | | | |
| # | Result | $n_r$ | $n_t$ | Steps | Time (s) | Steps | Time (s) | Steps | Time (s) | Steps | Time (s) |
| 01 | Live | 14 | 3 | 8 | 1.08 | 5 | 0.00 | 5 | 0.00 | 5 | 0.00 |
| 02 | Dead | 14 | 3 | 8 | 0.98 | 10 | 0.00 | 7 | 0.00 | 5 | 0.00 |
| 03 | Live | 14 | 3 | 8 | 0.93 | 5 | 0.00 | 5 | 0.00 | 5 | 0.00 |
| 04 | Live | 30 | 2 | 15 | 1.20 | 4 | 0.00 | 4 | 0.00 | 4 | 0.00 |
| 05 | Live | 30 | 3 | 20 | 1.31 | 5 | 0.00 | 5 | 0.00 | 5 | 0.00 |
| 06 | Dead | 30 | 3 | 20 | 2.78 | 19 | 0.03 | 9 | 0.04 | 5 | 0.00 |
| 07 | Dead | 38 | 5 | 34 | 37.31 | 34 | 0.17 | 7 | 0.00 | 5 | 0.00 |
| 08 | Live | 46 | 5 | 33 | 1.78 | 5 | 0.00 | 5 | 0.00 | 5 | 0.00 |
| 09 | Dead | 38 | 6 | 37 | >60.00 | 37 | 0.26 | 14 | 0.25 | 7 | 0.01 |
| 10 | Dead | 38 | 7 | 42 | 4.23 | 42 | 0.02 | 2 | 0.00 | 2 | 0.00 |
| 11 | Dead | 62 | 2 | 27 | 17.60 | 26 | 3.30 | 15 | 3.30 | 3 | 0.00 |
| 12 | Dead | 62 | 4 | 39 | >60.00 | 40 | 1.70 | 20 | 9.60 | 8 | 0.19 |
| 13 | Dead | 62 | 4 | 39 | >60.00 | 40 | 1.30 | 20 | 11.00 | 8 | 0.12 |
| 14 | Live | 62 | 4 | 39 | 3.27 | 6 | 0.00 | 6 | 0.01 | 6 | 0.02 |
| 15 | Dead | 46 | 4 | 42 | >60.00 | 42 | 0.22 | 15 | 1.30 | 6 | 0.01 |
| 16 | Live | 62 | 5 | 50 | 5.33 | 5 | 0.00 | 5 | 0.00 | 5 | 0.01 |
| 17 | Live | 62 | 4 | 50 | 43.11 | 6 | 0.00 | 6 | 0.01 | 6 | 0.02 |
| 18 | Dead | 62 | 4 | 50 | >60.00 | 49 | 2.10 | 15 | 13.10 | 6 | 0.05 |
| 19 | Dead | 62 | 5 | 51 | >60.00 | 50 | 0.83 | 16 | 36.00 | 6 | 0.03 |
| 20 | Dead | 70 | 5 | 57 | >60.00 | 56 | 1.20 | - | >60.00 | 6 | 0.03 |

Table 2: Performance evaluation on the instances from [21]. The $n_r$ and $n_t$ columns show the number of routes and number of trains, respectively, indicating the size of the problem instance. The “Ticks MIP alg.” column refers to the algorithm in [21] and relays the worst-case results reported in that paper. For algorithms 1, 2, and 3, the “Steps” columns contains to the number of transitions that have been added to the SAT formula when the algorithm terminates, and “Time” column reports the running time in seconds.

We compare our algorithms to the *reported* performance from [21] instead of executing a complete comparison test on the same machine because the source code and the closed-source commercial MIP solver library used in [21] were not available to us. This means that the comparison is not completely valid, as differences in hardware, solver libraries, and the heuristic for number of transitions may influence the result. Note also that the algorithm in [21] computes more than just the DEAD/LIVE status, also suggesting a set of safe locations to park trains for recovery operations. Still, we claim that the marked difference in steps and running times between [21] and our Algorithm 3 shows that our improvements to a large extent solve the challenges from that paper's conclusion, and opens the possibility for adequate performance also on larger-scale problem instances. The implementation source code for algorithms 1, 2, 3, and the problem instances, are available online[1].

## 5 Conclusions and Future Work

We have demonstrated how to use an incremental SAT solver to determine whether a railway system is bound for a deadlocked state. By allowing multiple trains to move by multiple elementary routes for every transition, we greatly lower the number of transitions required to show that the system is *not* bound-for-deadlock. By adding a constraint that forces all resource allocations to happen immediately after a conflicting resource has been released, we greatly lower the maximum number of steps where trains can make meaningful progress, allowing us to show that that the system *is* bound-for-deadlock in fewer steps. Together, these features result in a smaller propositional logic formula to be solved by the SAT solver because fewer transitions are required to decide the DEAD/LIVE answer, resulting in a much more efficient algorithm for the online railway deadlock detection problem. Efficient deadlock detection can be useful in non-autonomous decision support systems for manual railway dispatching or in semi-autonomous or autonomous controllers for railway dispatching or train driving.

We plan to further investigate the possible integration between non-complete/heuristic analysis algorithms and complete analysis algorithms for specific critical properties, in the context of autonomous railway operations. Studying the transition system described in this paper is not only relevant for deadlock detection in manual or autonomous railway control, but the efficiency gains described in this paper may also be applied to other analysis tasks in railway design and operations. We plan to investigate the grouping together adjacent of infrastructure elements and/or sets of trains to hide parts of the problem instances, and then performing counter-example-guided abstraction refinement. This might further reduce the sizes of the SAT instances required in complete planning algorithms for railways applications.

## Acknowledgements

We thank Veronica Dal Sasso for supplying benchmark problem instances and helping with the interpretation of the data. We thank Carlo Mannino for comments on the ideas and comparison of our approach with the literature on railway deadlocks.

## References

[1] Maurice H. ter Beek, Arne Borälv, Alessandro Fantechi, Alessio Ferrari, Stefania Gnesi, Christer Löfving & Franco Mazzanti (2019): *Adopting Formal Methods in an Industrial Setting: The Railways Case*. In

[1] `https://github.com/luteberget/deadlockrail`

Maurice H. ter Beek, Annabelle McIver & José N. Oliveira, editors: *Formal Methods - The Next 30 Years, FM 2019, Proceedings*, *Lecture Notes in Computer Science* 11800, Springer, pp. 762–772, doi:10.1007/978-3-030-30942-8_46.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman & Yunshan Zhu (2003): *Bounded model checking*. *Adv. Comput.* 58, pp. 117–148, doi:10.1016/S0065-2458(03)58003-2.

[3] Armin Biere, Katalin Fazekas, Mathias Fleury & Maximillian Heisinger (2020): *CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020*. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo & Martin Suda, editors: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, *Department of Computer Science Report Series B* B-2020-1, University of Helsinki, pp. 51–53.

[4] Armin Biere, Marijn Heule, Hans van Maaren & Toby Walsh, editors (2009): *Handbook of Satisfiability*. *Frontiers in Artificial Intelligence and Applications* 185, IOS Press.

[5] Magnus Björk (2011): *Successful SAT Encoding Techniques*. *J. Satisf. Boolean Model. Comput.* 7(4), pp. 189–201, doi:10.3233/sat190085.

[6] Ralf Borndörfer, Torsten Klug, Leonardo Lamorgese, Carlo Mannino, Markus Reuther & Thomas Schlechte, editors (2018): *Handbook of Optimization in the Railway Industry*. Springer, doi:10.1007/978-3-319-72153-8.

[7] Niklas Eén & Niklas Sörensson (2003): *Temporal induction by incremental SAT solving*. *Electron. Notes Theor. Comput. Sci.* 89(4), pp. 543–560, doi:10.1016/S1571-0661(05)82542-3.

[8] Wei Fang, Shengxiang Yang & Xin Yao (2015): *A Survey on Problem Models and Solution Approaches to Rescheduling in Railway Networks*. *IEEE Trans. Intell. Transp. Syst.* 16(6), pp. 2997–3016, doi:10.1109/TITS.2015.2446985.

[9] Alessandro Fantechi (2013): *Twenty-Five Years of Formal Methods and Railways: What Next?* In Steve Counsell & Manuel Núñez, editors: *Software Engineering and Formal Methods - SEFM 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8368, Springer, pp. 167–183, doi:10.1007/978-3-319-05032-4_13.

[10] M.P. Fanti, A. Giua & C. Seatzu (2006): *Monitor design for colored Petri nets: An application to deadlock prevention in railway networks*. *Control Engineering Practice* 14(10), pp. 1231–1247, doi:10.1016/j.conengprac.2006.02.007. The Seventh Workshop On Discrete Event Systems (WODES2004).

[11] Koji Hasebe, Mitsuaki Tsuji & Kazuhiko Kato (2017): *Deadlock Detection in the Scheduling of Last-Mile Transportation Using Model Checking*. In: *15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, DASC/PiCom/DataCom/CyberSciTech 2017*, IEEE Computer Society, pp. 423–430, doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.84.

[12] Keijo Heljanko (2001): *Bounded Reachability Checking with Process Semantics*. In Kim Guldstrand Larsen & Mogens Nielsen, editors: *CONCUR 2001 - Concurrency Theory, Proceedings*, *Lecture Notes in Computer Science* 2154, Springer, pp. 218–232, doi:10.1007/3-540-44685-0_15.

[13] Linh Vu Hong, Anne E. Haxthausen & Jan Peleska (2017): *Formal modelling and verification of interlocking systems featuring sequential release*. *Sci. Comput. Program.* 133, pp. 91–115, doi:10.1016/j.scico.2016.05.010.

[14] Feng Li, Jiuh-Biing Sheu & Zi-You Gao (2014): *Deadlock analysis, prevention and train optimal travel mechanism in single-track railway system*. *Transportation Research Part B: Methodological* 68, pp. 385–414, doi:10.1016/j.trb.2014.06.014.

[15] Quan Lu, Maged M. Dessouky & Robert C. Leachman (2004): *Modeling train movements through complex rail networks*. *ACM Trans. Model. Comput. Simul.* 14(1), pp. 48–75, doi:10.1145/974734.974737.

[16] Bjørnar Luteberget, Koen Claessen, Christian Johansen & Martin Steffen (2021): *SAT modulo discrete event simulation applied to railway design capacity analysis*. *Formal Methods in System Design*, pp. 1–35, doi:10.1007/s10703-021-00368-2.

[17] Franco Mazzanti, Alessio Ferrari & Giorgio Oronzo Spagnolo (2016): *Experiments in Formal Modelling of a Deadlock Avoidance Algorithm for a CBTC System*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, *Lecture Notes in Computer Science* 9953, pp. 297–314, doi:10.1007/978-3-319-47169-3_22.

[18] Franco Mazzanti, Giorgio Oronzo Spagnolo, Simone Della Longa & Alessio Ferrari (2014): *Deadlock Avoidance in Train Scheduling: A Model Checking Approach*. In Frédéric Lang & Francesco Flammini, editors: *Formal Methods for Industrial Critical Systems, Proceedings*, *Lecture Notes in Computer Science* 8718, Springer, pp. 109–123, doi:10.1007/978-3-319-10702-8_8.

[19] Jörn Pachl (2007): *Avoiding Deadlocks in Synchronous Railway Simulations*. In: *2nd International Seminar on Railway Operations Modelling and Analysis. 2007, Hannover, Germany*, doi:10.24355/dbbs.084-200704030200-0. Available at https://publikationsserver.tu-braunschweig.de/receive/dbbs_mods_00020794.

[20] Jussi Rintanen, Keijo Heljanko & Ilkka Niemelä (2006): *Planning as satisfiability: parallel plans and algorithms for plan search*. *Artif. Intell.* 170(12-13), pp. 1031–1080, doi:10.1016/j.artint.2006.08.002.

[21] Veronica Dal Sasso, Leonardo Lamorgese, Carlo Mannino, Andrea Onofri & Paolo Ventura (2021): *The Tick Formulation for deadlock detection and avoidance in railways traffic control*. *J. Rail Transp. Plan. Manag.* 17, p. 100239, doi:10.1016/j.jrtpm.2021.100239.

[22] Mary Sheeran, Satnam Singh & Gunnar Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In Warren A. Hunt Jr. & Steven D. Johnson, editors: *Formal Methods in Computer-Aided Design, FMCAD 2000, Proceedings*, *Lecture Notes in Computer Science* 1954, Springer, pp. 108–125, doi:10.1007/3-540-40922-X_8.

[23] Carsten Sinz (2005): *Towards an Optimal CNF Encoding of Boolean Cardinality Constraints*. In Peter van Beek, editor: *Principles and Practice of Constraint Programming - CP 2005*, *Lecture Notes in Computer Science* 3709, Springer, pp. 827–831, doi:10.1007/11564751_73.

[24] Yakir Vizel, Georg Weissenbacher & Sharad Malik (2015): *Boolean Satisfiability Solvers and Their Applications in Model Checking*. *Proc. IEEE* 103(11), pp. 2021–2035, doi:10.1109/JPROC.2015.2455034.