Preprint

SAT modulo Discrete Event Simulation Applied to Railway Design Capacity Analysis

Bjørnar Luteberget · Koen Claessen · Christian Johansen · Martin Steffen

2021-03-18

Abstract This paper proposes a new method of combining SAT with discrete event simulation. This new integration proved useful for designing a solver for capacity analysis in early phase railway construction design.

Railway capacity is complex to define and analyze, and existing tools and methods used in practice require comprehensive models of the railway network and its timetables. Design engineers working within the limited scope of construction projects report that only ad-hoc, experience-based methods of capacity analysis are available to them. Designs often have subtle capacity pitfalls which are discovered too late, only when network-wide timetables are made – there is a mismatch between the scope of construction projects and the scope of capacity analysis, as currently practiced.

We suggest a language for capacity specifications suited for construction projects, expressing properties such as running time, train frequency, overtaking and crossing. Such specifications can be used as contracts in the interface between construction projects and network-wide capacity analysis. We show how these properties can be verified fully automatically by building a special-purpose solver which splits the problem into two: an abstracted SAT-based dispatch planning, and a continuous-domain dynamics with timing constraints evaluated using discrete event simulation. The two components communicate in a

The first author was partially supported by the project RailCons, — *Automated Methods and Tools for Ensuring Consistency of Railway Designs,* with number 248714 funded by the Norwegian Research Council and Railcomplete AS.

Bjørnar Luteberget
SINTEF Digital AS, Norway
E-mail: bjornar.luteberget@sintef.no

Koen Claessen
Chalmers University, Sweden
E-mail: koen@chalmers.se

Christian Johansen
Norwegian University of Science and Technology, Norway
E-mail: christian.johansen@ntnu.no

Martin Steffen
University of Oslo, Norway
E-mail: msteffen@ifi.uio.no

CEGAR loop (counterexample-guided abstraction refinement). This architecture is beneficial because it clearly distinguishes the combinatorial choices on the one hand from continuous calculations on the other, so that the simulation can be extended by relevant details as needed. We describe how loops in the infrastructure can be handled to eliminate repeating dispatch plans, and use case studies based on data from existing infrastructure and ongoing construction projects to show that our method is fast enough at relevant scales to provide agile verification in a design setting. Similar SAT modulo discrete event simulation combinations could also be useful elsewhere where one or both of these methods are already applicable such as in bioinformatics or hardware/software verification.

Keywords SAT, discrete event simulation, capacity analysis, specification language, railway designs, SMT

1 Introduction

1.1 SAT modulo Discrete Event Simulation

Solvers for the Boolean satisfiability problem (SAT) are today efficiently solving many problems of practical significance, especially in electronic hardware and software verification. For problems that are inexpressible or inefficiently solved in propositional logic, satisfiability modulo theory (SMT) solvers allow richer expressions, and leverage the algorithms in SAT solvers by combining them with specialized solvers for other logics, called *theories* (introductory texts can be found in e.g. [43,2,3]).

The theories supported by SMT solvers are typically related to theorem proving (uninterpreted functions, integer arithmetic, real arithmetic), program verification (bitvectors, arrays, algebraic data types, strings), or mathematical programming (linear and non-linear arithmetic). In principle, however, the SMT solver architecture can be applied to any problem where solving a Boolean abstraction of the problem can contribute a major part of the solution. A theory solver must, as a minimum requirement, be able to assess an assignment to the Boolean abstraction of the given problem and decide whether that assignment makes the formula satisfiable in the theory's interpretation of the Boolean values, and if so, produce an assignment for the theory variables. To get an *efficient* solver, the following features of a theory solver are beneficial:

- Explanations: when a Boolean assignment is found unsatisfiable in the theory, the theory solver must be able to express a Boolean constraint that excludes that assignment. In the worst case, this constraint is just the negation of the whole assignment, but more succinct explanations give better pruning of the search tree.
- Early pruning: if the theory solver can check consistency of a *partial* Boolean assignment, while the SAT solver is building its variable assignment, the theory can prune the search tree more efficiently.
- Incrementality: if the theory solver can cheaply maintain the consistency check when adding new parts of the partial assignment, the early pruning is more efficient.
- Deduction: if the theory solver can take a partial Boolean assignment and deduce other Boolean assignments that necessarily follow from the current assignment, this also prunes the search tree without needing to keep track of more Boolean constraints.

Even if one cannot fulfill these criteria to get the optimal symbiosis effect of the SAT solver with the theory, the SMT method can still be worthwhile as a way to build solvers for hard problems as long as the Boolean abstraction in some way represents essential features

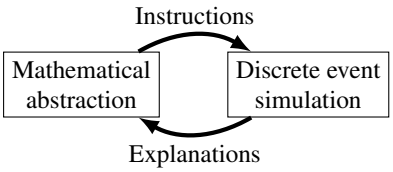

Fig. 1: Extending an exact but abstract mathematical model with a richer simulation model. A solution to the mathematical model is evaluated by sending instructions to a discrete event simulator. The simulator sends back explanations of shortcomings in the instructions in terms of the mathematical abstraction.

of the solution space. Even some problems that are well-suited for propositional logic, such as hardware model checking problems, can benefit from starting from a Boolean abstraction of the problem. The technique of counterexample-guided abstraction refinement (CEGAR) [7] can make solving model checking problems more efficient by abstracting away parts of the system being verified, and only adding it to the propositional formula when it is found to be necessary for drawing a conclusion.

In most engineering disciplines, exact mathematical models and solutions such as the ones used in SMT solvers are used for some analysis tasks, while other tasks are better suited for simulation. Computer-based simulation tools offer a different trade-off: they typically work by mimicking a system locally, going forward in time, and can more easily account for complex behavior without requiring a complete global description. In return, simulation can only tell where a system ends up after starting from a specific state, and not what to do to end up with a specific outcome. Discrete event simulation (DES) is a specific simulation technique used in many application areas for simulating multiple processes or agents that interact with a common environment [16, 13]. For example in transportation, each vehicle is modeled as a process, traffic light systems are processes, and these interact though shared resources in a common environment.

To produce a detailed model of moving vehicles at small time scales, acceleration and braking effects are essential. Since acceleration and braking are described mathematically by second order derivatives, the resulting mathematical model is necessarily non-linear and large systems of this type cannot be solved efficiently in general. An approach for solving complex vehicle movement problems can be to consider acceleration and braking to be instantaneous (SMT using linear arithmetic), or even abstracting away all continuous variables and representing only the sequencing of operations as a SAT problem (e.g., which vehicle first reserves a shared resource). Using DES as a theory solver for evaluating the abstracted solutions in detail can still lead to good solver performance for the overall problem if the abstracted problem represents the essential choices to be made, and the output from DES can be analyzed to produce explanations for failures as Boolean constraints (see Fig. 1).

This paper shows how to use discrete event simulation to extend exact mathematical modeling in propositional logic, focusing on applying SAT modulo DES to solve the problem of *capacity analysis* in railway construction planning. The SAT abstraction is used for handling the *planning* part of our problem, whereas the DES is used for evaluating *timing constraints* taking realistic dynamics of trains into account. We thus separate our problem into a discrete domain and a continuous domain part, and SAT modulo DES proves to be especially well suited for such a separation.

1.2 Motivations from railway construction planning

Railway design and construction planning are old engineering disciplines with long-standing traditions. Demands for the highest safety, compatibility with existing infrastructure and practices, and high investment costs, make railway engineering a conservative domain. The design process of railways is in practice highly sequential, leading to the known advantages and disadvantages of so-called waterfall process models.

Waterfall-style design processes require that high-level specifications can be written up-front and implemented afterwards without feedback from the implementation process back to the high-level specifications. This also means that verification and validation in waterfall-style design processes is confined to the scope of each separate design activity, or destined to have little hope of improving the design when weaknesses are uncovered.

Unfounded design assumptions made in the early process stages have been known to trickle all the way down to the final stages and require new rounds of design starting from the top, a process which typically takes several years.

These negative effects are typically mitigated by:

a) Re-using proven design concepts, i.e., doing something the same way as somewhere else, where it has already turned out to work well.
b) Allowing sizable margins, e.g., planning the track with more than enough space for safety distances so that it is highly likely that control system engineers will later be able to come up with a safe and performant design.

These mitigations exploit railway engineers' traditions, experiences, and cross-discipline knowledge, which in turn contributes to making the engineering community slow-moving and conservative.

Modern construction practice, however, expects and demands optimization. When space requirements, performance requirements, and costs are squeezed to the limits, the tradition-based railway engineering approach lacks the methods to accurately reason about the limitations of the finished system on the basis of partially finished designs.

1.3 Capacity analysis at railway design time

The planning and engineering of a railway control system has safety as primary requirement. Safety is ensured through the so-called signaling principles, and detailed requirements have been put in place for station layouts, controller implementations, and operation procedures.

Secondary to safety, the notion of performance and capacity of a railway control system remains more elusive. The *capacity of a railway control system*, and thus of railway infrastructure in general, is hard to define precisely (see [19,1,27]). Any capacity measure will necessarily make assumptions about the operation of the railway. One can say that the railway infrastructure does not have an inherent capacity, only capacity for specific use cases. A fully accurate assessment of capacity can only be made under a fully specified timetable, meaning that every train's arrival and departure times at all stations in the network must be known. This makes for a highly coupled analysis, as constructing an actual timetable requires bringing together details about infrastructure, rolling stock, transportation demands, and crew schedules. Systematic capacity analysis for railways is typically performed on the scale of national railway networks, using comprehensive input on infrastructure and timetables, and only after planning and engineering has produced a final design. Moreover, the widely used methods and tools for capacity analysis are heavy-duty methods, consisting of

complicated simulations, and require specialized knowledge. Thus they are unsuitable for more agile design-time verification of railway stations.

For construction projects and control system engineering, it is not feasible to use a fully specified timetable for verifying that the control system will be able to provide the required capacity, because (1) detailed timetabling and capacity analysis takes too much effort and specialized knowledge, and is usually saved for later stages of design, and (2) the design of a control system cannot or should not depend too heavily on other parts of the network, as these parts may also change in the future.

On the other hand, analytical approaches to railway capacity analysis, such as queueing network theory and maximum flow calculations, abstract away too much information to be useful for low-level infrastructure design engineering. Simplifying assumptions, which can be suitable for network-scale capacity analysis, such as instantaneous speed changes, or fixed traveling times between different locations, are usually not suitable for infrastructure design. Specifically, disregarding the discrete allocation logic of the interlocking system and the position and velocities of individual trains makes these methods unsuitable for analysis of signalling design. The detailed optimization of signal and detector locations needs to account for a detailed model of train dynamics and control system behavior exactly because higher-level analysis requires this assumption of local optimization to the simplified behaviors used in network-global analysis.

As none of these techniques are particularly well-suited for agile design-time verification of railway stations, engineers working on construction projects usually rely on informal, vague, or even non-existent capacity specifications, and need to make ad-hoc analyses of how the control system might provide this capacity.

Using *agile verification* of high-level properties from the beginning of a design project, and in every step of the process, allows engineers to better see the consequences of each decision and immediately uncover errors and shortcomings that would otherwise be discovered only months or years later.

Our goal is to develop a verification technique and tool to help engineers specify capacity properties *at design time* and to check these automatically. To be agile, the tool needs to

a) have reasonable response times so that the verification can be run on the fly as the design is being updated by an engineer working in a drafting CAD application, and
b) keep the required input to the minimum of information needed to verify relevant properties.

This style of verification gives engineers immediate feedback on their design decisions while requiring small amounts of specification and verification work.

Problem definition

We address the following problem: in the context of designing the layout and control systems for railway stations, does the station infrastructure have the *capacity* to handle the amount of trains and the desired traveling times to provide adequate service in transportation of goods and passengers? (See exact definition below.)

As an example, consider the question of crossing trains on a railway station. Fig. 2 shows two sequences of movements which result in such a crossing. Details of the railway design (such as signal placement, detector placement, correct allocation and freeing of resources, track lengths, train lengths, etc.) may render this scenario infeasible, or cause it to take an unacceptably long time.

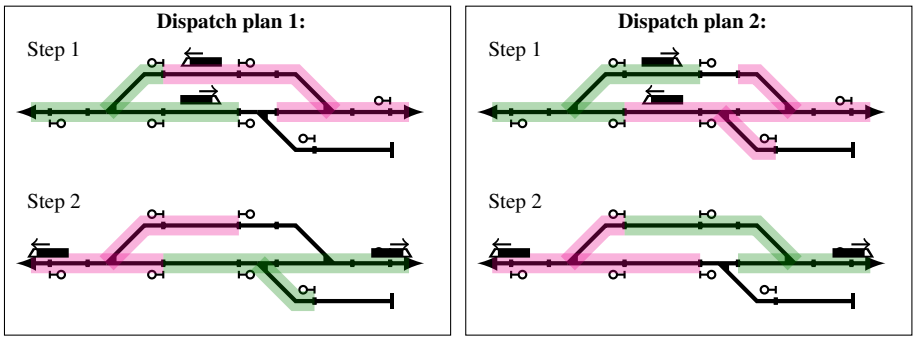

Fig. 2: Two alternative dispatch plans for achieving a crossing of two trains on a small example station. The green areas show track segments which are currently allocated to a train going from left to right, while the pink areas show track segments which are currently allocated to a train going from right to left. After each step of the plan, segments behind the front of each train are freed if they are not still occupied by the length of the train. In the last step, the trains are exiting the model boundary and all their allocated sections are freed.

We consider **the low-level railway infrastructure capacity verification problem**, which we define as follows:

> Given a railway station track plan, including signaling components, rolling stock dynamic characteristics, and a performance/capacity specification, verify whether the specification can be satisfied and find a dispatch plan as a witness to prove it.

Solving this problem subsumes the following railway infrastructure design activities:

a) Low-level **running time** analysis – verify that getting from point A to point B can be achieved within a given time.
b) Low-level **schedulability** analysis – verify that the infrastructure can handle given frequencies of trains arriving and departing at a station, and simultaneous opportunities for crossing, parking, loading, etc.
c) **Combinations** of the above – verify running time requirements on schedulable operations.

1.4 SAT modulo DES applied to low-level railway infrastructure capacity verification

We suggest a formalization of capacity requirements as a set of operational scenarios involving a set of trains, a set of locations to visit, and a set of timing constraints.

Verification in this domain can in principle be encoded into the SMT [3, 8, 36] or PDDL+ [14] languages, essentially resulting in a SAT modulo non-linear real arithmetic problem [15, 24]. The PDDL+ language has been designed to express planning problems in mixed discrete/continuous domains. We were able to use the SMTPlan+ solver to solve only the most trivial test cases in less than one second, which is the time bound that we consider a reasonable running-time constraint for agile verification. Translating the railway capacity problem as a whole into a PDDL+ instance makes SMTPlan+ create a high number of planning steps for any combination of train control events and interlocking events. There are

several SMT solvers that can handle SAT modulo non-linear real arithmetic problems [9, 17, 10], but we found these solvers insufficiently scalable for real-world problem sizes. Using SMT solvers directly suffers from the same problem of having a high number of planning steps (some improvements can be made, such as making train driver choices implicit in constraints on the relation between velocity, distance and time). Moreover, assuming that the train dynamics are given by an explicit expression over real numbers may be too simplistic for some engineering applications. We would like to have the opportunity to extend the dynamics equations in the future using, e.g., numerical integration.

We have developed a verification tool chain that uses a CEGAR loop [7] between a SAT-based planning tool that works on a discrete abstraction of the control system commands, and a discrete event simulation engine (DES) [42] that calculates detailed continuous results for a specific plan, taking the physics of moving trains into account.

The SAT-based planner uses bounded model checking (BMC) [4] where time is reduced to a series of partially ordered actions with unknown durations, and the choice of actions are the available commands in the control system. The DES component verifies the continuous time/space results given the Boolean decisions of control system commands, and adds new SAT constraints excluding unsatisfactory solutions.

The separation of discrete and continuous domains also has the advantage that the simulation component can be extended to handle more complex models, such as engine power curves, tunnel air resistance, curve rolling resistance, train weight distribution, etc., without affecting the planning logic or its computational complexity.

We have developed a prototype DES engine for our specific simulation needs, but the flexibility of the proposed method of SAT modulo DES allows for other (off-the-shelve) DES engines to be used, maybe adapted or with an interface to handle the interactions with the SAT engine. The application domain and running time requirements would dictate the choice of specific DES method and engine.

1.5 Contributions and organization of the paper

This paper extends our work from [31] in the following ways: *(i)* We provide an expanded introduction to railway control systems and dynamics in Sec. 2, to make the work self-contained. *(iii)* We have added new results for handling safety zones in Sec. 3.2. *(iv)* We have added new results for handling loops in infrastructure and eliminating repetition in dispatch plans in Sec. 3.3 and Sec. 3.4. *(ii)* We give more detailed descriptions of the implementation of discrete event simulation in Sec. 4.

The paper is organized as follows. Sec. 2 contains an overview of the railway design process and the principles for analysis of these designs. In Sec. 2.2 we present a language for capacity specifications, together with examples of how it can be used in construction projects. Sec. 3 describes the tool chain and the solver architecture that we propose to verify performance properties in an agile verification style, integrated in the construction project workflow. We present in detail in Sec. 3.2 how the planner component of our solver is implemented. The simulator component is described in Sec. 4. Sec. 5 contains performance evaluations in a set of relevant case studies. Sec. 6 discusses further related work and presents our conclusions.

We have tested our method and tool on practical examples from existing infrastructure and ongoing construction projects in collaboration with engineers from Railcomplete AS.

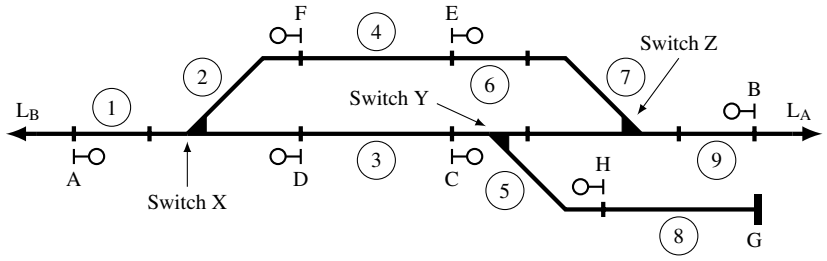

Fig. 3: Schematic presentation of railway infrastructure. Signals are shown as symbols labeled A-G, though G is not an actual signal but marks the end of a siding track (implicitly signalling stop). The arrows, labeled L_A and L_B, indicate that the track continues, but outside the scope of the model. Short vertical bars indicate detectors. The detector arrangement forms sections of the track labeled with circled numbers 1-9.

Elementary route	Entry signal	Exit signal	Switch position	Track segments	Conflicts
AC	A	C	X right	1, 2, 3	AE, BD, HD, DL_B , FL_B
AE	A	E	X left	1, 2, 4	AC, BF, DL_B, FL_B
CG	C	G	Y right	5, 8	CL_A, BD, HD
CL_A	C	L_A	Y left, Z left	5, 7, 9	CG, EL_A, BD, BF, HD
EL_A	E	L_A	Z right	6, 7, 9	CL_A, BD, BF
BD	B	D	Z left	9, 7, 5, 3	AC, CG, CL_A, EL_A, BF
BF	B	F	Z right	9, 7, 6, 4	AE, CL_A, EL_A, BD
HD	H	D	Y right	5, 3	AC, CG, CL_A, BD
DL_B	D	L_B	X right	2, 1	AC, AE, H_B
FL_B	F	L_B	X left	2, 1	AC, AE, D_B

Table 1: Example of a *tabular interlocking specification*, showing available routes and their conditions for the infrastructure in Fig. 3.

The tool can be found online[1] and can be used either standalone or integrated in a railway design framework to provide on-the-fly capacity analysis.

2 Railway construction capacity analysis problem background

The signalling and interlocking design problem for a railway station takes the track plan as input, typically containing tracks, switches and platforms, and produces the following artifacts:

- Track and trackside component layout, describing the locations of tracks, switches, signals and detectors (see Fig. 3).
- Interlocking specifications, describing the requirements for the logic of the control system (see Table 1).

[1] Source code: `https://github.com/koengit/trainspotting` and Documentation: `https://luteberget.github.io/rollingdocs`

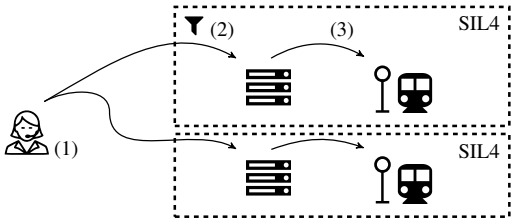

Fig. 4: A dispatcher (1) requests routes from the interlocking control system. The interlocking decides whether to accept the command (2), and signals the resulting movement authority to the train driver (3). The control system itself is responsible for the safety of the resulting movements. A railway construction project often concerns the design of only one or a few of these control systems, so the scope of the interlocking acts as a module boundary in the larger railway network.

These design artifacts are the subject of verification, i.e., they constitute the *model*. Ensuring performance in the context of a construction project consists of verifying *properties* describing a set of trains moving on the tracks and the goals which need to be accomplished by these movements. The design should (1) ensure safe movement while also (2) fulfilling performance requirements. We describe each of these aspects in the sections below.

To verify performance properties, we need to find a sequence of trains and elementary routes for the train dispatcher, i.e., a *dispatch plan*, which, when executed under safety and correctness constraints (described in Sec. 2.1 below), satisfies the properties described in the performance requirements (detailed in Sec. 2.2 below).

Train movements along the railway are coordinated by a train dispatcher, whose task it is to choose which trains go where, and communicate this to the train drivers. The dispatcher uses a control system to perform this task, called the interlocking, which receives input from trackside train detectors and controls movable track elements and signals (see Fig. 4).

2.1 Safe and correct train movements

Low-level analysis of train movements covers a wide range of constraints given by the track layout, the control system, and operational procedures. The following subsections give an overview of these constraints, divided into four classes. See [38] for a more in-depth description of railway operation principles.

2.1.1 Physical infrastructure

Trains travel on a network of railway tracks with physical properties such as length, gradient, curvature, etc. Tracks branch off using *switches*, whose *setting* determines where the train goes. Detectors on the track are used by the control system to determine whether track segments are occupied. The physical infrastructure also determines the *sight areas*: the set of locations where a train receives information from a given signal. The different visual appearances that a signal can have are called *aspects*. For example, the “proceed aspect” displays a green light to indicate that the train should proceed, though the aspect may not necessarily be (only) one green light and there are more and quite more complex aspects

for signals compared to the traffic lights used in road traffic. The exact list of visual aspects depends on national regulations and does not matter very much for the content of the work.

Tracks which are not connected to switches end in either (1) a *dead end* (e.g., the location labeled G in Fig. 3) where the train cannot travel further, or (2) a *model boundary* where trains can appear and disappear from the model (e.g. the locations labeled L_A and L_B in Fig. 3).

2.1.2 Interlocking: allocation of resources

The safety-critical control systems for railway infrastructure are called *interlockings*. An interlocking takes requests for activating routes from a dispatcher. When a route is activated, switches are moved into correct positions and signals are set to show the proceed aspect. The interlocking is also responsible for assuring that activating the route, i.e., allowing the train to travel the route, is safe.

The safety of train movements is ensured through the following requirements:

a) Routes require the exclusive allocation of **track segments**, so that two routes which use some of the same track segments cannot be activated at the same time. Routes must be allocated as a unit, i.e., all segments must be free at the time of allocation. However, track segments may be de-allocated (and then used in other routes) as soon as the train has passed a segment. The whole of the route being allocated is called a **elementary route**, while the parts that can be de-allocated separately are called **partial routes**.
b) **Switches** need to be in the correct position for the train to travel along the route. Also, the switches must be locked, so that they cannot accidentally be moved while the train is traveling, and detectors on the switch must report that the switch is actually locked in the correct position.
c) A **safety zone** (also called overlap) beyond the end of the route must be vacant, but not necessarily exclusively allocated, i.e., two safety zones may share track segments. The safety zone is released after a given time which is long enough that it is unlikely that the train is still moving forward. This timeout is calculated based on the length of the route.
d) Routes which pass through switches require specific track elements to cover any potential movements into the route path. This is known as **flank protection**, and can typically be provided by signals, switches, or other objects.
e) **Signals** can only show the "proceed" aspect when it is the starting point for a currently active route; in all other states, the signal must show the "stop" aspect. Distant signals, i.e., additional signals showing information about the next upcoming route, must give information consistent with the upcoming signal.

These constraints are explicitly expressed for a given railway station through the interlocking specification, which is an artifact of the design process.

Avoiding collisions by exclusive use of resources is the responsibility of the interlocking, which takes requests from the dispatcher for activating **elementary routes**. An elementary route is the smallest unit of resources that can be allocated to a train, see Fig. 5. Route activation can be modeled as a process which executes as follows:

a) Wait for all **required resources**, such as track segments and switches, to be free. Resources required by a route are typically any resource in the train path (or sometimes outside of it), which ensure that all movements are performed at a safe distance from each other.

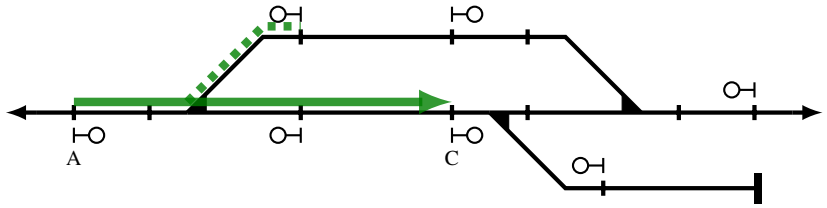

Fig. 5: Elementary route AC from signal A to the adjacent signal C. The thick line indicates parts of the track on the train's path which are reserved for this movement, and the dashed lines indicate parts of the track outside the path which are also exclusively allocated.

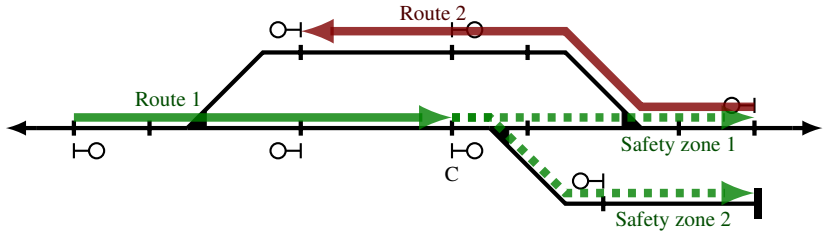

Fig. 6: Elementary route 1, ending in signal C, can protect trains from overrunning the signal by allocating one of the safety zones (shown as safety zone 1 and 2). In some situations, safety zone 1 might be preferred so that the switch following signal C is in the correct position for letting the train in route 1 proceed quickly. However, allocating safety zone 1 blocks route 2 from use. So in other situations, safety zone 2 might be preferred, for example for two trains to concurrently enter a station. Some control systems may allow one safety zone to be replaced by another after the route has been allocated.

b) **Movable elements** (e.g. switches) must be set to correct positions. If they are not, start a sub-process which moves the element into place, and wait for this process to finish before proceeding.
c) **Signals** are then set to show the "proceed" aspect to the train when the above steps are finished. When the front of the train has passed the signal, it is immediately reset to show the "stop" aspect.
d) A **release** process is started, which waits for the train to finish using the allocated resources (i.e., to travel over them) and frees them when this has happened.

Influence of safety zones on capacity

The safety zone, as described above, is a set of track sections and switches allocated together with a route to ensure that slightly overrunning a signal showing the stop aspect is safe (see Fig. 6). Different manufacturers and national regulations have various ways of specifying how a safety zone is released and how alternative safety zones are implemented. The main variations are:

a) The safety zone from a route end point persists until a route is allocated from the end point. This can be problematic if the safety zone blocks other traffic or if the train is changing directions and not proceeding past the end point. The following two methods are the usual mitigations for these problems.

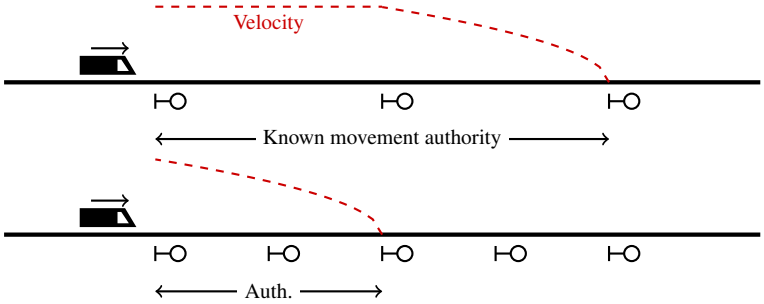

Fig. 7: Signal information only carries across two signals (so-called *distant signals*).

b) The safety zone is released after a pre-set time. This time should be long enough so that the probability that the train is still running towards the end point is very low.
c) The safety zone it not released, but can be replaced by another safety zone from the same route end point. This method is called *swinging overlap* in the United Kingdom.

2.1.3 Communication constraints

After movement has been allowed by the control system, the driver must be informed of this fact. When a route is activated, a train inside the sight area of the route's entry signal reads the signal's message that movement authority is given. The train driver may then drive the train forward until the next signal.

The following types of signalling systems are common in railways:

a) Traditional signaling with trackside lamps. Communication is limited by how many different aspects the lamps can show. To avoid high-speed trains slowing down at every signal, several consecutive elementary routes can be signaled in advance using so-called distant signals.
b) Automatic train protection systems (ATP) work similarly to signals, but may give more information. Many ATP systems communicate information through magnets or short-range radio at specific locations on the track, corresponding to a signal sight area of zero length.
c) The European Rail Traffic Management System (ERTMS) currently being implemented in many European countries replaces lamp signals with trackside marker boards, and uses long-range radio for communication. This effectively removes the communication constraint, as the radio can be used to update any train's movement authority at any time.

The amount of information that can be transmitted to the train drivers through the signaling puts a constraint on how many consecutive routes ahead it is possible to give movement authority for. Traditional signaling can typically show information about either one or two routes, but some countries have extended to information about three or four consecutive routes. It is also common to extend the information given by signals using track-side electronic communication. See Fig. 7.

2.1.4 Laws of motion

Trains move within the limits of given maximum acceleration and braking power, so train drivers need to plan ahead for braking so that the train respects its given movement authority and speed restrictions at all times.

The speed increase from v_0 to v over a time interval Δt is limited by the train's maximum acceleration a:

$$v - v_0 \leq a\Delta t \ .$$

However, when there is a more restrictive speed restriction ahead, the driver must start braking in time to meet the restriction. A signal showing the 'stop' aspect can be treated as a speed restriction of zero. Since speed restrictions change with time, the driver must re-evaluate their actions whenever new information is received.

A train has the following constraint on its velocity v for each restriction,

$$v^2 - v_i^2 \leq 2bs_i \ ,$$

where v_i is the maximum allowed speed, s_i is the distance to the location where the restriction starts, and b is the maximum deceleration achieved by braking. These restrictions are given as (1) constant maximum velocity restrictions given by signs beside the track, or (2) dynamical velocity restriction given by the distance to the next stop signal (i.e., the length of the movement authority).

2.2 Performance requirements specifications language

To capture typical performance and capacity requirements in construction projects, we define an **operational scenario** $S = (V, M, C)$ as follows:

a) A set of **vehicle types** V, each defined by a length l, a maximum velocity v_{max}, a maximum acceleration a, and a maximum braking deceleration b.
b) A set of **movements** M, each defined by a vehicle type and an ordered sequence of visits. Each visit q is a set of alternative locations $\{l_i\}$ and an optional dwelling time t_d.
c) A set of **timing constraints** C, each constraint consisting of two visits q_a, q_b, and an optional numerical constraint t_c on the maximum time between visit q_a and q_b. The two visits can come from different movements. q_a is required to occur before q_b, and q_b is required to occur within a time t_c after q_a:

$$t_{q_a} \leq t_{q_b}, \quad t_{q_a} + t_c \geq t_{q_b}$$

If the time constraint t_c is omitted, the visits are only required to be ordered (effectively $t_c = \infty$).

To demonstrate how an operational scenario captures requirements of railway construction projects, we give some examples using the syntax of the file format used in our tool[2]. First, we define the following vehicle types:

```
vehicle passengertrain length 220.0 accel 1.0 brake 0.9 maxspeed 55.0
vehicle freighttrain length 850.0 accel 0.5 brake 0.5 maxspeed 20.0
```

[2] For details of the input file formats, see `https://luteberget.github.io/rollingdocs/usage.html`

The following types of performance requirements are typical examples from our collaboration with railway engineers from Railcomplete AS:

a) **Running time**: expresses an expectation of how long it should take for a train to travel between two locations. To specify this, we simply require that a train visits some location `b1` and later visits some other location `b2`. A timing constraint of 90.0s between these visits sets the running time requirement.

```
movement passengertrain {
  visit #a [b1]; visit #b [b2] }
timing a <90.0 b
```

The notation `timing a <90.0 b` indicates that a should happen before b, with b happening no later than 90.0 seconds after a.

b) **Train frequency**: a train station processes a set of trains arriving and departing with a fixed frequency. On a two-track station, we exemplify a sequence of four trains and their relative departure times as:

```
movement passengertrain {
  visit [b1]
  visit [platform1,platform2] wait 60.0
  visit #e1 [b2] }
// ...3 more trains with visits e2, e3, e4.
timing e1 <90.0 e2
timing e2 <90.0 e3
timing e3 <90.0 e4
```

c) **Overtaking**: trains traveling in the same direction can be reordered. For example, we specify a passenger train traveling from `b1` to `b2`, and a freight train with the same visits. Timing constraints ensure that the passenger train enters first while the freight train exits first.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement freighttrain {
  visit #g_in [b1]; visit #g_out [b2] }
timing p_in < g_in
timing g_out < p_out
```

d) **Crossing**: trains traveling in *opposite directions* can visit this station simultaneously. This example is similar to the previous one, but the freight train now travels in the opposite direction, and the timing constraints require that the trains are inside the model simultaneously.

```
movement passengertrain {
  visit #p_in [b1]; visit #p_out [b2] }
movement freighttrain {
  visit #g_in [b2]; visit #g_out [b1] }
timing p_in < g_out
timing g_in < p_out
```

Similar specifications, and combinations of such specifications, are relevant in most railway construction projects. Since we typically only need to refer to locations such as model boundaries and loading/unloading locations, these specifications are not tied to a specific design, and can often be re-used even when the design of the station changes drastically.

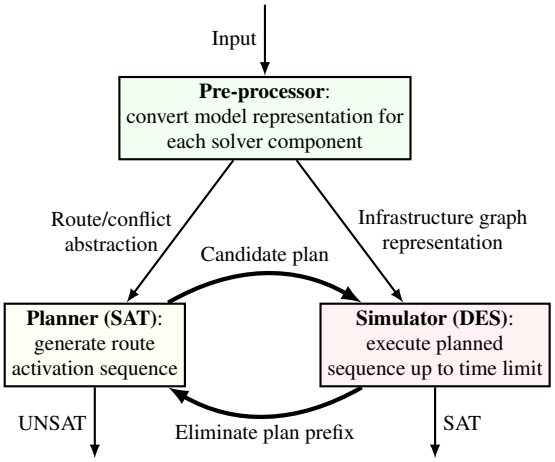

Fig. 8: Conceptual diagram of our CEGAR architecture. Infrastructure, routes, train types, and movement specifications are transformed into (1) the planner's abstract representation, containing only elementary routes and train lengths, and (2) the detailed graph representation used in the simulator component.

3 Agile capacity analysis in railway designs

In this section, we describe our approach to verifying capacity properties using the formalization of capacity given in the previous section.

3.1 SAT modulo DES tool-chain for capacity analysis in railway designs

We developed a CEGAR-style tool which exploits the limited number of control system commands to make an abstraction of the planning problem, see Fig. 8. Towards the end of the section this tool-chain is extended to include handling of loops and repetitions culminating in the diagram from Fig. 15 on page 27. Our overall tool chain for solving the low-level railway infrastructure capacity verification is outlined in Fig. 9 along with information flow between the components. The manual, source code and test cases are available online[3]. The tool uses the MiniSAT v2.2.0 solver.

The tool is complementary to other verification techniques in railway design, such as static layout verification [33,32,30], static interlocking verification [21,32], interlocking program verification [5], and timetable analysis [20].

The following input documents are used:

a) **Operational scenarios** defining the performance properties to verify. Examples are given in Sec. 2.2.
b) **Infrastructure** given in the railML format [35,41]. In our case studies, railML was generated using the RailCOMPLETE software, a plugin for the widely used AutoCAD

[3] `https://luteberget.github.io/rollingdocs` and `https://github.com/koengit/trainspotting`

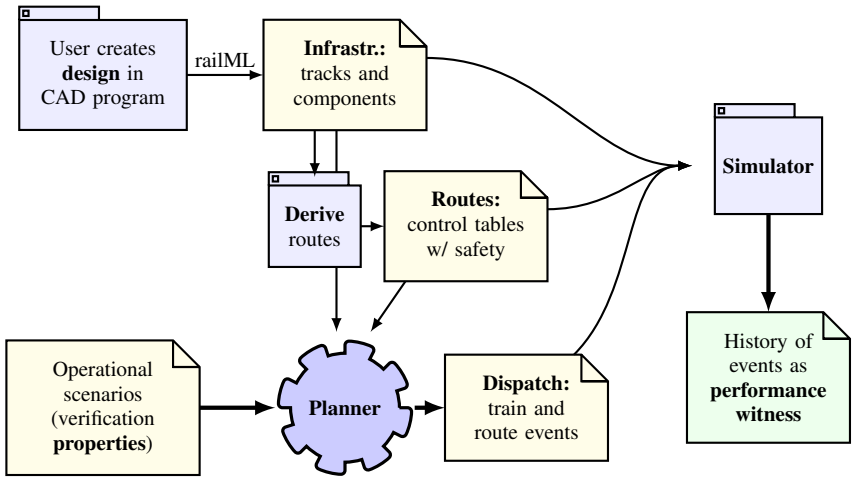

Fig. 9: Capacity verification tool chain overview. Yellow boxes represent input documents. Note that only infrastructure and operational scenarios are strictly required – interlocking tables can be derived, and dispatch plans can be synthesized. Blue boxes represent programs. The green box represents the output document from the simulator, which is a history of events which is the witness that proves the performance requirements.

drafting software. In this way the model is taken directly from the engineers' drafting program with no additional model preparation needed.

c) **Elementary routes** (*optional*), given in a custom format which is compatible with the upcoming railML interlocking format. Although subject to design, a decent guess of the content can be straight-forwardly derived from the infrastructure by listing resources on paths between adjacent signals, so this input is optional.
d) **Dispatch plans** (*optional*) corresponding to each operational scenario. The verification tool can produce dispatch plans fulfilling the performance specification, so this input is optional.

The verification program works by first running the *planner*, which generates and solves a SAT problem representing a discrete abstraction of the overall verification problem to produce candidate dispatch plans. The simulator component, which evaluates the time consumption of plans, reports which parts of the plan fail the timing constraints, and the negation of this partial plan is added to the SAT instance. Since the timing calculations are path dependent, we use the part of the plan starting from the beginning and going up to the step where the timing specification violation occurs. This way of refining the abstraction can cause performance problems when many different choices are possible early in the plan, and the timing violation can only be found near the end of the plan, as demonstrated in Sec. 5. Finding a way to make more precise refinements could be necessary for larger problem instances.

An advantage of the separation of planner and simulator is that each component can be used separately. *The planner alone* may be used to enumerate different possibilities for train movements, which might be used in an operational testing situation. *The simulator alone* may be used to debug the execution of a specific dispatch plan to examine performance deficiencies, and educationally for demonstrating the workings of the railway system. Put

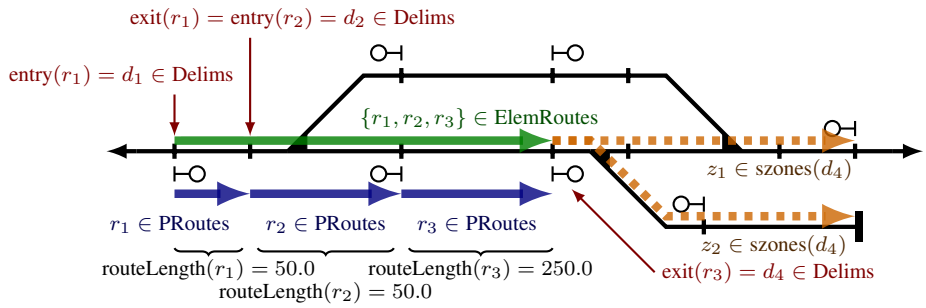

Fig. 10: Abstracted infrastructure input used for converting a part of the infrastructure into the planning SAT problem. The elementary route AC (see Fig. 5) consists of three partial routes, r_1, r_2, r_3, connected through their shared delimiters (exit(r_1) = entry(r_2), etc.). The exit signal of the elementary route, shown here as delimiter d_4 (signal C in Fig. 5), can have two alternative safety zones z_1, z_2 (see Fig. 6).

together, the two components provide automated verification, which is the main goal of our efforts. It would also, in principle, be possible to use one of the commercial simulation packages, such as OpenTrack or RailSys, provided that all input and simulation control can be given though a programmable interface (API).

3.2 Dispatch Planning using SAT

The planner solves the abstracted discrete planning problem of finding a dispatch plan, i.e., determining a sequence of trains and elementary routes which ensure that the trains visit locations according to the movements specification.

We encode an instance of the abstracted planning problem into an instance of the Boolean satisfiability problem (SAT). We consider the problem a model checking problem, and use the technique of bounded model checking (BMC) to unroll the transition relation of the system for a number of k steps, expressing state and transitions in propositional logic.

Using BMC for planning works by asserting the existence of a plan, so that satisfiability of the corresponding SAT instance shows that the performance requirements can be fulfilled and gives an example plan for it. When unsatisfiable, we are ensured that there is no plan within the number of steps k. We pick a bound k based on practical considerations (letting k be twice the number of trains was sufficient in our case studies), and we are not interested in searching for plans longer than that. The SAT instance is built incrementally by solving first with only one step and then adding new steps when the formula is found to be unsatisfiable, adding up to k steps if necessary.

The planner needs only to work on an abstracted infrastructure containing information about routes and their connectedness and conflicts, and does not directly make considerations such as setting the positions of switches. The smallest pieces of the infrastructure that the planner needs to know about are the *partial routes*, which are the units of de-allocation for elementary routes. The following input structures are used when converting the infrastructure into the planning SAT problem (see also examples in Fig. 10):

– A set of partial routes, PRoutes.

- A set of route delimiters, Delims. Each such delimiter element represents either a signal or a detector in the original infrastructure model.
- A set of safety zones, SZones.
- Each partial route's entry and exit delimiters, which take the null value at the boundaries of the model. I.e. $\text{entry}(r) = \text{null}$ if partial route r starts from the model boundary and $\text{exit}(r) = \text{null}$ it ends at the model boundary,

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

- Each route delimiter's choice of safety zones,

$$\text{szones} : \text{Delim} \to \mathcal{P}(\text{SZones}).$$

- A set of pairs of routes which are conflicting, i.e. they are not allowed to be used at the same time,

$$\text{RConflicts} \subseteq \text{PRoutes} \times \text{PRoutes},$$

  and a set of safety zones conflicting with partial routes,

$$\text{SZConflicts} \subseteq \text{SZones} \times \text{PRoutes}.$$

- Each partial route's length,

$$\text{routeLength} : \text{PRoutes} \to \mathbb{R}.$$

- A set of elementary routes, each represented as a set of partial routes,

$$\text{ElemRoutes} \subseteq \mathcal{P}(\text{PRoutes}).$$

In addition to the infrastructure, we also need to represent the operational scenario. Here, the planner abstraction needs to know only about train visits as a set of partial routes containing the specified visit locations. Also, the ordering constraints on visits is needed, along with the length of the trains. The operational scenario is structured as follows when input to the planner:

- A set of trains, Trains.
- A set of specified visits, Visits.
- Each train's length,

$$\text{trainLength} : \text{Trains} \to \mathbb{R}.$$

- Each train's ordered list of specified visits,

$$\text{trainVisits} : \text{Trains} \to \text{Visits}^*,$$

  where a visit is a set of alternative partial routes, each of which contain at least one of the locations from the specifications defined in Sec. 2.2:

$$\text{visitRoutes} : \text{Visits} \to \mathcal{P}(\text{PRoutes}).$$

- An ordering of visits representing both the ordering given by timing constraints and the ordering given by the train's sequence of visits:

$$\text{VisitOrd} \subseteq \text{Visits} \times \text{Visits}.$$

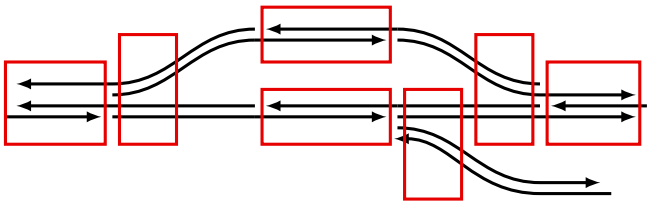

Fig. 11: The planner component takes an abstracted view of the railway infrastructure. Lines represent elementary routes with traveling direction given by the arrows. Boxes indicate routes in conflict, i.e. only one of them can be in use at a time.

The objective of the planning algorithm is to produce a set of *dispatch plans*, consisting of an ordered sequence of commands,

$$\text{Plans} = \text{Cmd}^*, \quad \text{Cmd} = \text{TrainCmd} \cup \text{RouteCmd},$$

where train commands describe trains entering the model from a boundary using an elementary route, i.e. TrainCmd $=$ Trains $\times$ RouteCmd. Route commands are given by a route and a safety zone: RouteCmd $=$ ElemRoutes $\times$ SZones.

The abstracted planning problem is encoded as a SAT instance by representing states, constraints on each state, and constraints on consecutive states. We use variables taking values from finite sets, and encode this into a SAT problem using a one-hot encoding.

State i of the system in the planner component is represented as:

a) Each partial route r has an **occupancy status** o_r^i which is either free ($o_r^i = \text{Free}$) or occupied by a specific train t ($o_r^i = t$).
b) Each route delimiter d has a **safety zone status** u_d^i which is either unused ($u_d^i = \text{None}$) or set to one of the delimiter's safety zones ($u_d^i = z$), where $z \in \text{szones}(d)$.
c) Each train t has a Boolean representing **appearance status** b_t^i, used to propagate to future states that a train has started (used in constraint C1 below).
d) Each visit v has a Boolean representing **required visits status** q_v^i, which is used to propagate to future states that a visit requirement has been fulfilled (used in constraint C6).
e) Each combination of route r and train t has a Boolean representing **deferred progress** $p_{r,t}^i$, used to propagate to future states that a train is not progressing, and must resolve the conflict in the future (used in constraint C9).

A dispatch plan is produced directly from the occupancy status o_r^i and safety zone choices u_r^i of states by taking the difference between consecutive states and then dispatching any trains and elementary routes (with safety zones) which become active from one state to the next. Note that many elementary routes may be allocated to a train in a single planning step, and this (in most instances) greatly reduces the total number of planning steps required.

Constraints are applied to each state and each pair of consecutive states to ensure that:

– The plan is viable for execution (i.e., **correctness**):
(C1) Each train takes one valid, contiguous path.
(C2) Safety zones are active at the end of each train's current path.
(C3) Conflicting routes are not activated simultaneously.
(C4) Elementary route are allocated as a unit.

(C5) Partial routes are deallocated only after a train has fully passed over them.
– The **plan fulfills capacity specifications**:
(C6) Trains perform their specified visits.
(C7) Visits happen in specified order.
– Equivalent solutions are eliminated (for increased **solver performance**):
(C8) Routes are deallocated immediately after the train has fully passed over them.
(C9) A train's path is extended as far as possible in the current time step, unless hindered by a conflicting train (i.e., we want maximal progress in each step).

Equivalent plans, which result in the same trains traversing the same paths and conflicting in the same locations, should have the same representation so that enumeration of different plans produces meaningful alternatives. For example, the two dispatch plans for crossing shown in Fig. 2 are the only two alternatives given by the planner for this operational scenario. See Fig. 12 for other dispatch plans which fulfill the correctness constraints (C1-7) but which do not have maximal progress in each state.

The implementation of each of these constraints as propositional logic statements is described below. Constraints apply separately to all states i unless noted otherwise.

3.2.1 Train path (C1)

At most one route is taken by a train in a single state. First, ensure that only one route from a given start delimiter may be taken at any time:

$$\forall t \in \text{Trains} : \forall d \in \text{Delims} \cup \{\text{null}\} : \text{atMostOne}(\left\{ o_r^i = t \mid \text{entry}(r) = d \right\}) .$$

We use a standard sequential encoding to encode atMostOne and other similar constraints, as explained in e.g. [44]. Note that entry delimiters for all routes entering from a model boundary share the same null value, so that this constraint also excludes plans where a single train appears in several positions at once. Each train should only enter the plan once, thus if the appearance variable has become true then it will stay true for all subsequent states:

$$\forall t \in \text{Trains} : b_t^i \Rightarrow b_t^{i+1}.$$

A train's appearance status changes when an entry boundary route is allocated:

$$\forall t \in \text{Trains} : \forall r \in \{r \in \text{PRoutes} \mid \text{entry}(r) = \text{null}\} : \left(o_r^i \neq t \wedge o_r^{i+1} = t \right) \Rightarrow (\neg b_t^i \wedge b_t^{i+1}).$$

Routes which are not entry 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:

$$\forall t \in \text{Trains} : \forall a \in \{a \in \text{PRoutes} \mid \text{entry}(a) \neq \text{null}\} :$$
$$\left(o_a^i \neq t \wedge o_a^{i+1} = t \right) \Rightarrow \bigvee \left\{ o_b^{i+1} = t \mid b \in \text{PRoutes}, \text{entry}(a) = \text{exit}(b) \right\} .$$

Note that this constraint ensures that the trains' allocation to routes *locally* forms a path in the graph of routes. In the presence of cycles, this constraint does not rule out cyclic allocations disjoint from the rest of the train's path. This problem is handled separately in Sec. 3.3 below.

3.2.2 Safety zones (C2)

A partial route delimiter needs a safety zone to be active only when the delimiter is the end point of a train movement:

$$\forall d \in \text{Delims} : \left(\bigvee \{o_r \neq \text{Free} \mid \text{exit}(r) = d\} \wedge \bigwedge \{o_r = \text{Free} \mid \text{entry}(r) = d\} \right)$$

$$\Rightarrow (u_d^i \neq \text{None}).$$

Safety zones are also involved in (C3) and (C9) below.

3.2.3 Resource conflicts (C3)

Any two routes which require the same resources cannot both be allocated in the same state.

$$\forall (a, b) \in \text{RConflicts} : o_a^i = \text{Free} \vee o_b^i = \text{Free}.$$

Safety zones also have conflicts with routes:

$$\forall d \in \text{Delims} : \forall (u, r) \in \{(u, r) \in \text{SZConflicts} \mid u \in \text{szones}(d) \wedge \text{exit}(r) = d\} :$$

$$u_d^i \neq z \vee o_r^i = \text{Free}.$$

3.2.4 Partial release (C4)

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

$$\forall t \in \text{Trains} : \forall x \in \text{ElemRoutes} : \text{allEqual}(\left\{ o_r^i \neq t \wedge o_r^{i+1} = t \mid r \in x \right\}).$$

3.2.5 Deallocation (C5, C8)

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

$$\forall t \in \text{Trains} : \forall r \in \text{PRoutes} : (o_r^i = t) \Rightarrow \left((o_r^{i+1} \neq t) \Leftrightarrow \text{freeable}_t^i(r, \text{trainLength}(t)) \right).$$

Note that the bidirectional implication sign on the right hand side means that deallocation is both allowed (C5) and required (C8).

The freeable formulas are produced by the following recursive function:

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

$$\bigvee_{\substack{b \in \text{PRoutes} \\ \text{exit}(a) = \text{entry}(b)}} (o_i^b = t) \wedge \left(\textbf{if } l_b \geq l \textbf{ then } \top \textbf{ else } \text{freeable}_t^i(b, l - l_b) \right),$$

where $l_b = \text{routeLength}(b)$. 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.

3.2.6 Visits (C6, C7)

The fulfillment of a visit constraint is indicated in the system state by the required visits status q_v^i, for each of the visit requirements $v \in \text{Visits}$. Like the train's appearance status b_t^i described above, the visit constraint needs only be fulfilled once:

$$\forall v \in \text{Visits} : q_v^i \Rightarrow q_v^{i+1}.$$

A visit v is only fulfilled when a train is allocated to one of the partial routes given by $\text{visitRoutes}(v)$:

$$\forall t \in \text{Trains} : v \in \text{trainVisits}(t) : \left(\neg q_v^i \wedge q_v^{i+1}\right) \Rightarrow \bigvee \left\{ o_r^{i+1} = t \mid r \in \text{visitRoutes}(v) \right\}.$$

Finally, visits must happen in the correct order:

$$\forall (a,b) \in \text{VisitOrd} : q_b^i \Rightarrow q_a^i.$$

For the last state i, q_v^i is given as a retractable assumption to the solver, which forces all visits to be fulfilled before the plan ends.

Note that nothing prevents a train which has already fulfilled a visit constraint from later using routes that could fulfill the visit constraint again. Visit constraints may be fulfilled by any of these route allocations. For acyclic infrastructure, this is fact is usually not relevant since a train cannot return to a previous location. See also the notion of repetition defined in Sec. 3.3 below.

3.2.7 Forced progress (C9)

In addition to the constraints on allocation and freeing required to produce a valid plan, we also add constraints which force each train to proceed further along a path forward unless there is a conflict. Routes ahead are either allocated, or the train is deferred p:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall a \in \text{PRoutes} : \\ &(o_a^i = t) \Rightarrow p_{t,a}^i \vee \bigvee \left\{ (o_b^i = t) \mid b \in \text{PRoutes}, \text{entry}(b) = \text{exit}(a) \right\}.\end{aligned}$$

Deferred progress can only be resolved by freeing a conflicting route and then allocating it to the train in the following step:

$$\begin{aligned}&\forall t \in \text{Trains} : \forall r \in \text{PRoutes} : \\ &p_{t,r}^i \Rightarrow p_{t,r}^{i+1} \vee \bigvee \{ o_a^i \neq t \wedge o_a^{i+1} = t \wedge o_b^i \neq \text{Free} \\ &\qquad\qquad \mid (a,b) \in \text{RConflicts}, \ \text{exit(r)} = \text{entry}(a) \}.\end{aligned}$$

For the last state $i+1$, $\neg p_{t,r}^{i+1}$ is given as a retractable assumption to the solver, which forces the deferred progress to be resolved before the end plan ends. Note that it is not required that the conflicting trains are distinct.

Similarly, safety zones should only be changed from one value to another if this causes a route which was previously in conflict to be used:

$$\begin{aligned}&\forall d \in \text{Delims} : \forall z \in \text{szones}(d) : (u_d^i = z) \Rightarrow \\ &\qquad (u_d^{i+1} = \text{None}) \vee (u_d^{i+1} = z) \vee \bigvee \left\{ o_r^{i+1} \neq \text{Free} \mid (z,r) \in \text{SZConflicts} \right\}.\end{aligned}$$

3.3 Handling turning and loops

Many railway construction projects have only *acyclic* infrastructure, in the sense that trains enter from one side of the station and exit on the other side, and all paths from one side to the other are acyclic.

However, if the infrastructure has a same-directed cycle which can be allocated without conflicting with other routes, the constraints C1 above are insufficient to ensure train path consistency, see Fig. 13. The train path consistency constraints described in the previous section require each active route to have a route before it already being active. This works in the acyclic case, because the chain of routes always leads back to either a model boundary or a route already allocated in the previous step. With cyclic infrastructure, however, a sequence of routes can justify each other, which would lead to a train appearing out of nowhere. It is a known problem that expressing this kind of constraints in SAT can be very inefficient (see e.g. [28, 18]), and to handle same-directed cycles in the infrastructure, we add instead a refinement step around the SAT solver which searches each state for this kind of circular reasoning and adds a single constraint each time this situation appears.

The loop check procedure checks for each train t_i and for each state s_j, whether the set of routes R_i^j allocated to the train has any strongly connected components $\text{scc}_i^j \subseteq R_i^j$ with $|\text{scc}_i^j| > 1$, and in that case adds a new constraint to the SAT problem:

$$\bigvee \left\{ \neg(o_r^j = t_i^j) \mid r \in \text{scc}_i^j \right\} .$$

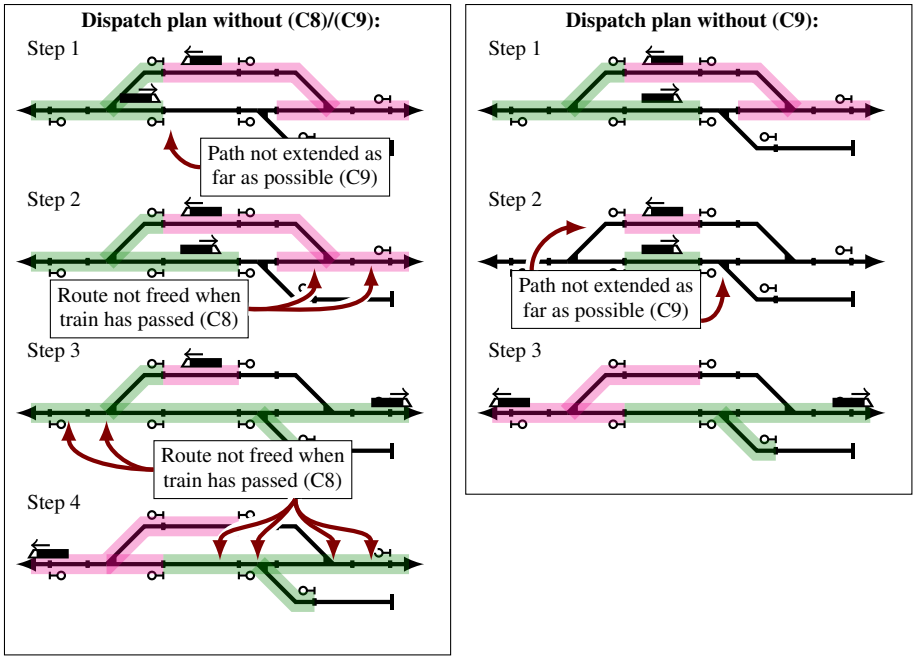

Fig. 12: Examples of dispatch plans which are correct plans (constraints (C1-7)), but which have better equivalent descriptions that allocate and deallocate as soon as possible. These plans do not fulfill constraints (C8) and (C9). Compare with plan 1 in Fig. 2.

Fixing these consistency errors gives valid plans even in the presence of same-directed infrastructure cycles, but even planning on infrastructure without cycles may cause repetition to appear in the dispatch plans. For example, at the end of a railway corridor, trains must be able to switch directions and go back to where they came from. In the description of dispatch planning above, if trains are allowed to stop and reverse their direction, the directed graph of routes becomes cyclic, and there is in principle an infinite number of different possible dispatch plans for any train movement.

Allowing trains to turn and/or allowing loops in the infrastructure, will lead to the bounded model checking procedure finding more and more solutions when increasing the number of steps. Most of these solutions will exhibit some amount of repetition in the movement of trains, and this makes them of little value to the railway engineer. We suggest some different solutions to this challenge below, roughly ordered by how complex the implementation would be and how much quality would be improved:

- **Unlimited:** it could be feasible to have no limit on turning of trains, and no limit on the use of loops in the infrastructure. Since the bounded model checking procedure will find the shortest plans first, they will often be the most valuable plans for the engineer, and the planner can be aborted when plans get too long and repetitive and as such are no longer valuable for the verification of the design. However, the fully automated verification tool would have to set a carefully considered upper bound on the number of plan steps.
- **Specified turning:** the specifications of the operational scenarios can be extended to include turning explicitly at visits. This increases the specification burden on the engineer, but ensures that there cannot be an unbounded number of distinct plans. However, it could also cause some plans to stay undetected if they require turning and the engineer did not think of it. Also, this method does not help the situation with loops in the infrastructure.
- **Bounded number of turns:** instead of writing out each turn explicitly, the capacity specifications could be extended to include an upper bound on the number of turns. The bound would have to be adjusted to balance running time and plan quality (low bound) with the possibility of detecting more complex plans (high bound).
- **State space repetition constraint:** to ensure that the whole state of the system does not repeat from one stage to another. This requires adding a constraint on each pair of

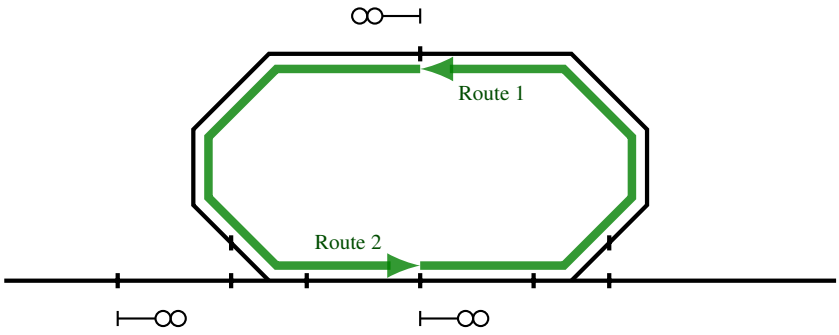

Fig. 13: Example of cyclic infrastructure. Here, to ensure train path consistency (C1), additional constraints are needed over the acyclic case. Route 1 and route 2 both provide each other's justification for a train appearing there, possibly making an error of circular reasoning.

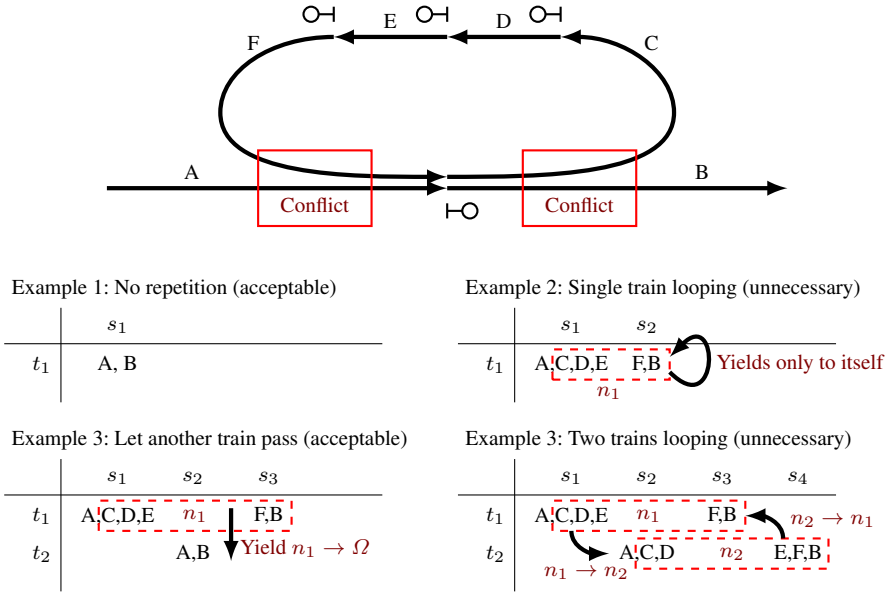

Fig. 14: Examples of repetition justification using yields, demonstrating acceptable and unnecessary repetitions. Each of the routes A, B, C, D, E, and F shown in the infrastructure route graph is long enough to contain each train completely. Examples use trains t_1, t_2 and states s_1, s_2, s_3, s_4. Repetitions are shown as red dashed boxes, and yields are shown as arrows between repetitions.

states, which could make the SAT instance significantly larger.

$$\bigwedge_{0 \le i < j < k} S_i \neq S_j.$$

Such constraints may also be added lazily, i.e. by incrementally adding the constraints only when they are violated in a SAT solution (see [11]). This constraint would eliminate the possibility for an infinite number of distinct plans, but could still cause unnecessary repetition locally, since repetition in one part of the model could be accompanied by progress in another part of the model.

– **Repetition filtering:** even when the state as a whole does not repeat, there may be sequences of allocation to a subset of trains which can be considered repeating. We would like a more domain-specific definition of repetition, based on a graph analysis of the dispatch plans produced. This can be implemented by rejecting solutions which exhibit such repetition. We define this more carefully in the section below.

As we find the last option to be the most complete solution requiring no change to the specifications, we describe its implementation here in more detail.

3.4 Filtering out unnecessary repetitions

We now define the notion of *unnecessary repetitions* and show how to identify them on a given dispatch plan. First, we define the notions of *yield* and *repetition*.

A train t_1 *yields* to another train t_2 if t_2 is occupying a route whose resources are needed for t_1 to proceed (thereby allowing t_1 to defer its progress as defined in constraint (C9), Sec. 3.2.7). More precisely, if t_2 occupies some route r_2 in state s, and t_1 allocates a route r_1 in state $s+1$, where r_1 conflicts with r_2, we say that t_1 yielded to t_2 in state s.

Now, consider a train t that enters the model from some model boundary and exits through another boundary by traveling a sequence of routes $r_1, \ldots, r_{m+1}$, which we call the train's *path*. For each pair of consecutive routes r_i, r_{i+1}, the exit signal of r_i is the same as the entry signal for r_{i+1} (described as constraint C1 in Sec. 3.2.1), which we call the delimiting signal $u_i = \mathrm{delim}(r_i, r_{i+1})$ between the routes r_i and r_{i+1}. We say that the train visits the sequence of signals $u_1, \ldots, u_m$ defined in this way.

A signal appearing several times in this sequence ($u_i = u_j$ with $i < j$) indicates a cycle in the train path. Let $s_a = \mathrm{alloc\_state}^t(r_i)$ be the state where route r_i starting in u_i is allocated to t, and let $s_b = \mathrm{alloc\_state}^t(r_{j-1})$ be the state where route r_{j-1} ending in u_j is allocated to t. We say that the train t *repeats* on the interval s_a to s_b and write $\mathrm{repeat}(t, s_a, s_b)$.

In most cases, we would like to disallow such repetitions, but there are two exceptions. Firstly, if the train fulfills a specified visit on the state interval s_a to s_b (see constraint (C6), Sec. 3.2.6), the repetition is acceptable. Secondly, if the train yields to another train in a state s_y such that $s_a \leq s_y \leq s_b$, we say that the yield *justifies* the repetition. For example, if a train goes into a siding track to allow another train to pass by, the first train could reverse into the main track again to proceed, thereby performing a repetition that is acceptable. See Fig. 14 for a few examples. However, if one repetition is justified by yielding to another train in a state which also has a repetition that is justified by yielding back to the first train, this does not make these repetitions acceptable. We would like to disallow such circular justifications, and we formalize this using the *yield justification graph*, $G = (V, E)$, defined as the directed graph where:

- The set of nodes N contains each repetition, $\mathrm{repeat}(t, s_a, s_b)$, and a special non-repetition node Ω.
- The set of edges E contains the edge $n_1 \to n_2$, where $n_1 = \mathrm{repeat}(t_1, s_a, s_b)$ and $n_2 = \mathrm{repeat}(t_2, s_c, s_d)$, whenever these nodes n_1, n_2 exist in N and t_1 yields to t_2 in a state s where $a \leq s \leq b$ and $c \leq s \leq d$.
  However, if $\mathrm{repeat}(\mathrm{t_1}, \mathrm{s_a}, \mathrm{s_b})$ exists, and t_1 yields to t_2 in state $s_a \leq s \leq s_b$, but there are no matching repetitions n_2, then the edge $n_1 \to \Omega$ is included instead.

We say that a repetition is acceptable if Ω is reachable from the repetition's corresponding node in the yield justification graph. A repetition that is not acceptable by these two criteria, is an *unnecessary repetition*, and we discard the candidate dispatch plan and add a new constraint to the SAT problem to disallow it using the relevant component of the yield justification graph. This adds another kind of abstraction refinement to our algorithm, see Fig. 15.

The methods for handling both loops and repetitions described here may cause performance problems on certain inputs. However, we have not encountered any real-world examples where this dominates the solver's performance.

4 Timing evaluation using Discrete Event Simulation

There are various well-known simulation approaches which are routinely and successfully used to analyze railway capacity. A simulation works by starting in a known state and, applying known input to the system, proceeds by executing imperative code to change the system

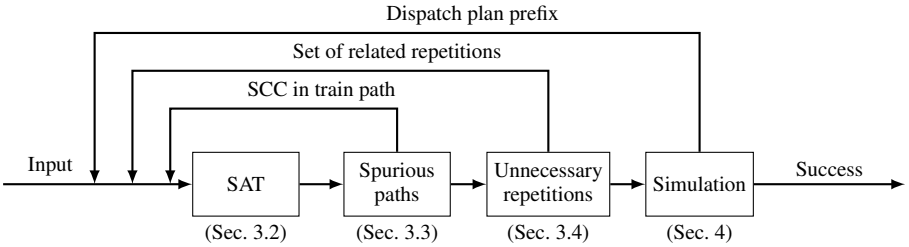

Fig. 15: Main algorithm for local capacity verification (extended from Fig. 8) with two more tests for handling loops and repetitions.

state and to register event handlers to processes. Deterministic simulation models can handle very complex models efficiently, but unlike a planning model, one cannot prescribe which state the simulation will end up in, only measure the outcome. Simulation methods are commonly used to develop and assess railway timetables. By introducing stochastic elements in the model and repeating the simulation a large number of times, also the robustness of a timetable can be analyzed (e.g., see [37]).

Discrete event simulation (DES) is based on the assumption that state changes happen only at discrete points in time, so that the simulation can progress efficiently by jumping from one point in time to the next where an event is scheduled. This can be made to work even for the continuous dynamics of train movements, as we assume that the dynamics of individual trains do not interact with each other directly: Trains are connected only through the control system, which has only discrete state changes. Each train acts separately on the information received from signals so far, and needs only to predict how long it will take to reach the next signal or sensor where it interacts with the control system.

In our tool architecture, the planner component works on an abstraction just detailed enough to ensure that trains end up where they are specified to go, and that the system does not enter a dead-lock state. This is the reason why the planning model must include safety zones, partial release, and the lengths of routes and trains – the sequences of routes and trains are represented precisely so that we know what to expect during the simulation. If it turns out that the planner's assumptions about where the trains end up do not work out correctly in the simulator, the correspondence between planning and simulation is broken, which may be a modeling error in the simulator or in the route specifications, e.g., if the switches are configured to turn in the wrong direction. Running the capacity verification assumes that the route specifications are correct, and this may be verified by other means (e.g., see [45]).

For the work in this paper, we have implemented a simulation system for railways containing main signals, detector, switches, routes, trains, partial release, safety zones and more.

4.1 The double-node graph

The input of railway infrastructure consists of *nodes*, representing locations on the tracks where transfer of information between the infrastructure and the train can happen. Objects include switches, detectors, signal sighting locations, and points of discrete changes in track properties such as radius and gradient. Nodes are connected by *edges* of a specified length. Edges are, on the one hand, undirected insofar they can be traveled in both directions. Directional information, on the other hand, is needed for a train to figure out which edges to

follow while still traveling in the same direction. But using a plain directed graph would require a *global* notion of direction, which is not compatible with cyclic infrastructure graphs where a train can travel back into the same track in the opposite direction. So, neither a plain directed nor undirected graph representation is fully adequate.

A more suitable data structure is the *double node graph* described in [34] where each node of a conventional graph is represented as two linked nodes representing each of the two sides for approaching each track location, see Fig. 16. A train reaching a node may only proceed by traveling on edges starting in the opposite node. Also, signals typically only apply in one of the traveling directions, so a train passing a pair of nodes only reacts to the objects that are located on the *exit side* of the node pair. This model allows for a *local* notion of directedness, and avoids deciding on a global direction concept such as up/down or outgoing/incoming often used in railway engineering. A global directionality requires considering special cases to handle railway networks where a train's up/down direction may change without the train reversing its direction, such as the balloon loop example which is commonly seen on tram lines, see Fig. 17.

4.2 Dispatch plans

From the SAT-based planner component described in Sec. 3.2 we extract **dispatch plans** as the external events given as input to the simulation, which involves events like starting a train, activating a route, or swinging a safety zone (i.e., replace an active safety zone by another). Note that the times at which these events happen are not given by the planner, only their *order* in time. All the rest of the simulation output is determined from these inputs. The inputs start processes in the simulator, which may in turn trigger other processes.

Each train's driver is represented as a process in the simulation system. We calculate a guaranteed minimum time until further action is required from the driver. This involves taking the minimum time until one of different relevant events happening, including that the train arrives at a new node or reaches maximum velocity (see Fig. 18)

4.3 Extensions and alternative simulators

In our simulation model, trains re-calculate braking curves analytically on every possibly relevant event. This makes for a high-performance system, but in real-world engineering there are other complexities that we do not yet handle in this system, such as:

- More complex signaling and automated train protection systems.

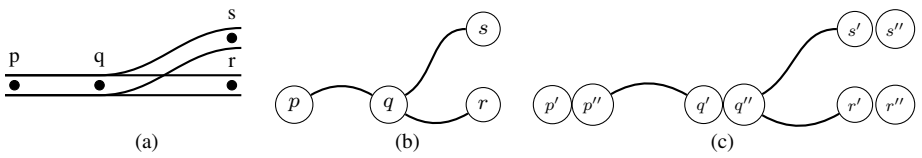

Fig. 16: (a) On the railway network, paths p-q-r and p-q-s exist, in both directions. (b) In a conventional undirected graph representation, there would also be a path r-q-s. (c) When the graph is extended to include two sides of each node, there is no longer a path r-q-s.

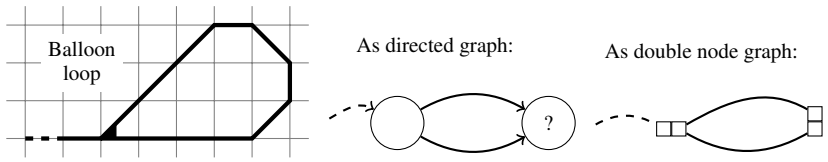

Fig. 17: The balloon loop infrastructure is an example where directionality of travel cannot be suitably captured as a directed graph.

- Local variations and details of infrastructure, such as the inner workings of components from different vendors performing various tasks like route allocation, de-allocation, safety zones, partial release, level crossings, etc.
- Train dynamics models using curve radius, gradient, air/tunnel resistance, weight distribution, etc.
- Stochastic variation in simulation output.

Our system can be extended with these features. Alternatively it would be possible to swap out our simulation module with a more comprehensive solution or a commercially available offering (see [37,29]), as long as this simulation program can be run in batch mode using the range of input described above. Also, implementing a discrete event simulation is most elegantly done through co-routines, such as in the SimPy[4] Python library, or though specialized languages for simulation such as ABS[5]. For the simple simulation system we have implemented, however, the number of distinct states in each type of process is small enough to managed by explicit state machine logic.

5 Case studies and performance

This section presents empirical evaluation of the running time of our tool on several examples. We picked representative examples based on real-world construction projects and

[4] See `https://en.wikipedia.org/wiki/SimPy`
[5] See `http://abs-models.org/`

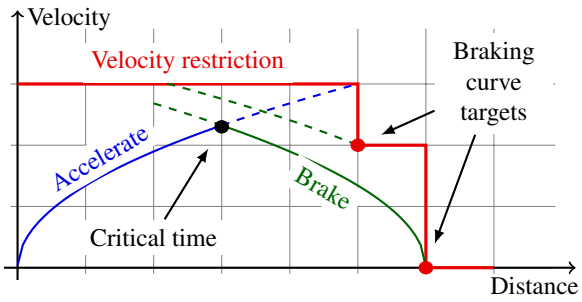

Fig. 18: The train driver's decision about when to accelerate/brake/coast happens at intersections between acceleration curves, braking curves and velocity restriction curves. In this example, the train can accelerate until the critical time where the acceleration intersects with the braking curve towards the second velocity restriction ahead (the first one is not critical).

Infrastructure	**Property**	**Result**	n_{DES}	t_{SAT}	t_{DES}	t_{total}
Simple (3 elem.)	Run.time	Sat.	1	0.00	0.00	0.00
	Crossing	Unsat.	0	0.00	0.00	0.00
Running ex. (12 elem.)	Run.time	Sat.	1	0.01	0.00	0.01
	Frequency	Sat.	1	0.01	0.00	0.01
	Overtaking 2	Sat.	1	0.00	0.00	0.01
	Overtaking 3	Unsat.	0	0.01	0.00	0.01
	Crossing 3	Unsat.	0	0.01	0.00	0.01
Kolbotn (BN) (56 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 4	Sat.	1	0.05	0.00	0.06
	Overtake 3	Unsat.	0	0.05	0.00	0.06
Eidsvoll (BN) (64 elem.)	Run. time	Sat.	2	0.01	0.00	0.02
	Overtake 2	Sat.	1	0.08	0.00	0.08
	Crossing 3	Sat.	1	0.04	0.00	0.04
	Crossing 4	Unsat.	0	0.21	0.00	0.21
Asker (BN) (170 elem.)	Overtaking 2	Sat.	1	0.20	0.00	0.21
	Overtaking 3	Unsat.	1	0.73	0.00	0.74
	Crossing 4	Sat.	0	0.75	0.00	0.77
Arna (CAD) (258 elem.)	Run. time	Sat.	1	0.02	0.00	0.04
	Overtaking 2	Sat.	1	0.50	0.00	0.51
	Overtaking 3	Sat.	1	1.43	0.00	1.45
	Crossing 4	Sat.	1	1.73	0.00	1.74
Gen. 3x3 (74 elem.)	High time	Sat.	1	0.01	0.00	0.01
	Low time	Unsat.	27	0.18	0.01	0.19
Gen. 4x4 (196 elem.)	High time	Sat.	1	0.01	0.00	0.03
	Low time	Unsat.	256	2.08	0.26	2.34
Gen. 5x5 (437 elem.)	High time	Sat.	1	0.06	0.00	0.09
	Low time	Unsat.	3125	38.89	4.35	43.24

Table 2: Verification performance on test cases, including Bane NOR (BN) and RailCOMPLETE (CAD) infrastructure models. The number of elementary routes (*elem.*) is shown for each infrastructure to indicate the model's size. n_{DES} is the number simulator runs, t_{SAT} the time in seconds spent in SAT solver, t_{DES} the time in seconds spent in DES, and t_{total} the total calculation time in seconds.

existing infrastructure. Verification performance on various test examples as well as real stations is presented in Table 2. The table shows the time spent in each solver component, and also the number of invocations n_{DES} of the simulator, which is very low in most of the practical cases. This supports our hypothesis that the chosen abstraction and CEGAR loop is efficient. The running example used in previous sections of this paper (see Fig. 3) is not too complex, having only 12 elementary routes (the 10 routes Table 1 plus two model boundary entry routes). Even so, this scale is still interesting for verification in practice, since there are many possible mistakes to uncover. The "*Running ex.*" infrastructure in Table 2 demonstrates the running times of the typical scenario specifications presented in Sec. 2.2. For example, "*Overtaking 2*" is the overtaking example from Sec. 2.2 where one train overtakes another, while "*Overtaking 3*" is a similar scenario where two trains arrive first, and then a third train overtakes them both. This is not possible on the example infrastructure without turning, as there are only two tracks available.

The Norwegian railway infrastructure manager Bane NOR has supplied a railML infrastructure model of the whole national railway network [40] from which we have extracted some more complex examples. Fig. 19 shows cut-outs from the visual representation of these models, i.e., the stations Kolbotn, Eidsvoll, and Asker were converted from the railML models. We have also tested against an infrastructure model from the Arna construction

project that uses the RailCOMPLETE CAD design software, a realistic use case for agile verification.

All of the realistic examples we constructed in our case study showed acceptable performance, with only two examples exceeding the target running time of 1 second and all of them finishing within 2 seconds. However, such low running times depends heavily on the fact that within the typical construction project there is not a high number of different paths that achieve very similar movements with very similar traveling times. For a property which has a high number of available dispatch plans (satisfying the abstracted requirements in the planner) where none of them satisfy the numerical timing constraints, the planner will have to try out all of them. To demonstrate this limitation of scalability in our method, we construct a set of examples where m stations each with n parallel tracks each are serially connected by a single track, show in the table as the infrastructures "*Gen 3x3*", "*Gen 4x4*", and "*Gen 5x5*". In this case, when a timing bound is slightly too small to be satisfiable, the planner will have to come up with n^m plans for timing evaluation.

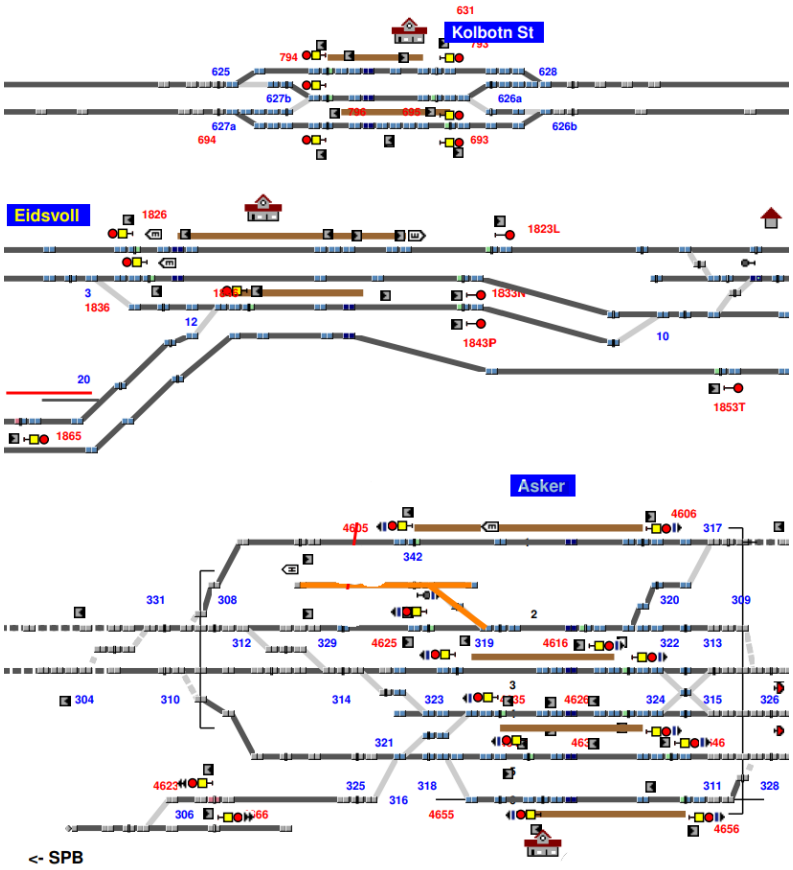

Fig. 19: Stations Kolbotn, Eidsvoll, and Asker from Bane NOR's model of the Norwegian national network [40].

These scaling scenarios are, however, outside the intended use case for the method described in this paper and instead puts us back in the realm of timetable analysis, where the infrastructure's influence on acceleration and braking behavior should be assumed to already be correctly designed at the station level, and/or a representative subset of possible plans can be analyzed stochastically. The *Gen* examples are included here to illustrate our method's limitations.

6 Conclusions and related work

We have proposed the combination of SAT with a discrete event simulation engine in a similar fashion as SAT modulo theories. This new combination differs from SMT mainly in the weaker form of interaction and the kind of help that the DES engine can give to the SAT solver. However, for particular problems, SAT modulo DES can be a successful combination allowing for fast running times on problems that are out of reach for both SAT or SMT. This is the case for the problem that we have investigated in this paper, coming from the railway design area. However, we think that SAT modulo DES combinations can prove fast for realistic problems in other domains where already one can find variants of both SAT and DES methods being used, such as in bioinformatics (e.g., haplotype inference or regulatory networks) or in hardware and software verification.

Using SAT modulo DES, we have proposed a solution and tool for automating the analysis of specific capacity aspects for railway infrastructure designs (defined in Sec. 1.3 as the *low-level railway infrastructure capacity verification* problem). The quite general types of capacity specifications that we can verify with our tool are those definable in the language proposed in Sec. 2.2.

The benefits for railway engineering that our suggested tool chain brings are twofold: (1) allow fully automated performance verification and (2) use minimal input documentation for the verification. Both of these aspects are needed in order to bring performance verification into the frequently changing early-stage design projects, thus avoiding the costly and time-consuming backtracking interactions required when later-stage analysis reveals unacceptable performance.

The control system design phase is lacking tools for rapid prototyping, namely tools anticipating to some extent the verification that is to be performed in later stages.

In response, we have demonstrated a control system design tool that can verify performance properties in the scope of a single project from high-level specifications by synthesizing schedules. Our work thus automates the following activities:

- Detailed **running time** analysis – verify the time required for getting from point A to point B, taking into account train dynamic characteristics, communication constraints, and control system logic and latency.
- Detailed **schedulability** analysis – verify frequency of trains arriving at a station, and simultaneous opportunities for crossing, parking, loading, etc.

Our introduction of SAT modulo DES to the railway design field opens up for new automation applications in the following sense: the level of detail supported by our tool is much greater than the traditional by-hand approaches for running time and schedulability analysis – and the amount of background data and work is much less than the whole-network stochastic operational analysis typically used in later-stage verification. To make our method approachable for engineers, the required input is the minimum of information needed to verify the relevant properties. For example, the specific paths each train takes through the station

is not an input, but different possibilities for realizing paths are explored by the verification procedure. This thus makes our method appropriate for early-stage design, where track lengths, topology, and component placement might be adjusted to achieve design goals, and engineers can in this way get feedback on design choices without requiring large efforts to repeat the verification.

Related work

Railway timetabling and capacity analysis has often been posed as a planning problem and solved using mixed integer programming (MIP) and similar approaches. Zwaneveld et al. [46] use integer programming on a problem closely related to our low-level railway infrastructure capacity verification. Isobe et al. [23] formulate a similar model in timed CSP, representing train locations, velocities, and control logic. Lamorgese et al. [26] describe a MIP-based approach for real-time decision support for dispatchers. Fedeli et al. [12] describe a MIP-based approach to line planning, i.e. making a selection from a set of alternatives lines that can satisfy high-level transportation demands. Our approach differs from all of the literature above in that our definition of the problem in this paper includes non-linear constraints on train dynamics (acceleration/braking power) and communication constraints (trains must slow down if they have not been informed of movement authority), which are relevant in construction projects but less relevant in high-level planning, real-time dispatching, and timetabling.

Many variations on discrete event simulation are used in railway dynamic analysis. A comprehensive account of object-oriented modeling and simulation of railway infrastructure is given in D. Hürlimann's Ph.D. thesis [22] (also based on M. Montigel's thesis [34]), which was later developed into the commerical simulation software OpenTrack. A similar approach presented in [25] uses futures and resource analysis support in the ABS programming language to simulate operational procedures.

In the planning literature, the PDDL+ language [14] has been introduced to capture mixed discrete/continuous planning problems such as the one studied in this paper. General-purpose solvers have recently been developed, using time domain discretization (DiNo [39]) or the SMT theory of non-linear real arithmetic (SMTPlan+ [6]).

Acknowledgments

We thank the engineers at Railcomplete AS, especially senior engineer Claus Feyling, for guidance on railway operations and design methodology.

References

1. Montserrat Abril, Federico Barber, Laura Paola Ingolotti, Miguel A. Salido, Pilar Tormos, and Antonio Luis Lova. An assessment of railway capacity. *Transportation Research Part E: Logistics and Transportation Review*, 44(5):774 – 806, 2008.
2. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
3. Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.

4. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.
5. Arne Borälv and Gunnar Stålmarck. Formal verification in railways. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 329–350. Springer, 1999.
6. Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full PDDL+ language into SMT. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *International Conference on Automated Planning and Scheduling, ICAPS 2016*, pages 79–87. AAAI Press, 2016.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2000.
8. Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
9. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
10. Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *26th Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.
11. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
12. Francesca Fedeli, Roberto Mancini, Carlo Mannino, Paolo Ofria, Gianpaolo Oriolo, Andrea Pacifici, and Veronica Piccialli. Optimal design of a regional railway service in italy. *J. Rail Transp. Plan. Manag.*, 7(4):308–319, 2017.
13. George S. Fishman. *Discrete-event simulation: modeling, programming, and analysis*. Springer Series in Operations Research. Springer, 2001.
14. Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.
15. Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.
16. Richard M Fujimoto. *Parallel and distributed simulation systems*. Wiley, 2000.
17. Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.
18. Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In Eduardo Fermé and João Leite, editors, *14th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 8761 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2014.
19. Ingo Arne Hansen and Jörn Pachl. *Railway Timetabling and Operations*. Eurailpress, 2014.
20. Steven S. Harrod. A tutorial on fundamental model structures for railway timetable optimization. *Surveys in Operations Research and Management Science*, 17(2):85 – 96, 2012.
21. Anne E. Haxthausen and Peter H. Østergaard. On the Use of Static Checking in the Verification of Interlocking Systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 266–278. Springer, 2016.
22. Daniel Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen*. PhD thesis, ETH Zurich, 2002.
23. Yoshinao Isobe, Faron Moller, Hoang Nga Nguyen, and Markus Roggenbach. Safety and Line Capacity in Railways – An Approach in Timed CSP. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *9th International Conference on Integrated Formal Methods (iFM)*, Lecture Notes in Computer Science, pages 54–68. Springer, 2012.
24. Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Comm. Computer Algebra*, 46(3/4):104–105, 2012.
25. Eduard Kamburjan and Reiner Hähnle. Uniform modeling of railway operations. In *Formal Techniques for Safety-Critical Systems FTSCS 2016*, volume 694 of *Communications in Computer and Information Science*, pages 55–71. Springer, 2016.
26. Leonardo Lamorgese and Carlo Mannino. An exact decomposition approach for the real-time train dispatching problem. *Oper. Res.*, 63(1):48–64, 2015.
27. Alex Landex. *Methods to estimate railway capacity and passenger delays*. PhD thesis, Technical University of Denmark (DTU), 2008.

28. Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
29. LUKS: Analysis of lines and junctions. Software web page: `http://www.via-con.de/development/luks`, 2018.
30. Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In Alessandro Cimatti and Marjan Sirjani, editors, *International Conference on Software Engineering and Formal Methods (SEFM)*, volume 10469 of *Lecture Notes in Computer Science*, pages 87–103. Springer International Publishing, 2017.
31. Bjørnar Luteberget, Koen Claessen, and Christian Johansen. Design-time railway capacity verification using SAT modulo discrete event simulation. In Nikolaj Bjørner and Arie Gurfinkel, editors, *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018.
32. Bjørnar Luteberget and Christian Johansen. Efficient verification of railway infrastructure designs against standard regulations. *Formal Methods in System Design*, 52(1):1–32, Feb 2018.
33. Bjørnar Luteberget, Christian Johansen, and Martin Steffen. Rule-based consistency checking of railway infrastructure designs. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods 2016*, volume 9681 of *Lecture Notes in Computer Science*, pages 491–507. Springer, 2016.
34. Markus Montigel. *Modellierung und Gewährleistung von Abhängigkeiten in Eisenbahnsicherungsanlagen*. PhD thesis, ETH Zurich, 1994.
35. Andrew Nash, Daniel Huerlimann, Jörg Schütte, and Vasco Paul Krauss. RailML — a standard data interface for railroad applications. In *Computers in Railways IX*, pages 233–240. WIT Press, 2004.
36. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
37. OpenTrack: Simulation of railway networks. Software web page: `http://www.opentrack.ch/`, 2018.
38. Jörn Pachl. *Railway Operation and Control*. VTD Rail Publishing, 2015.
39. Wiktor Mateusz Piotrowski, Maria Fox, Derek Long, Daniele Magazzeni, and Fabio Mercorio. Heuristic planning for PDDL+ domains. In Subbarao Kambhampati, editor, *International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 3213–3219. IJCAI/AAAI Press, 2016.
40. Bane NOR: Model of the Norwegian rail network. `http://www.banenor.no/en/startpage1/Market1/Model-of-the-national-rail-network/`, 2016.
41. railML. The XML interface for railway applications. Organization web page: `http://www.railml.org`, 2018.
42. Stewart Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, Inc., USA, 2004.
43. Roberto Sebastiani. Lazy satisability modulo theories. *J. Satisf. Boolean Model. Comput.*, 3(3-4):141–224, 2007.
44. Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.
45. Linh Hong Vu, Anne Elisabeth Haxthausen, and Jan Peleska. A domain-specific language for railway interlocking systems. In *10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 200–209. TU Braunschweig, 2014.
46. Peter J. Zwaneveld, Leo G. Kroon, and Stan P.M. van Hoesel. Routing trains through a railway station based on a node packing model. *European Journal of Operational Research*, 128(1):14 – 33, 2001.