Drawing with SAT: Four Methods and A Tool for Producing Railway Infrastructure Schematics

Bjørnar Luteberget[1] and Christian Johansen[2]

[1]SINTEF Digital, Oslo, Norway,
[2]Norwegian University of Science and Technology, Gjøvik, Norway

Abstract. Schematic drawings showing railway tracks and equipment are commonly used to visualize railway operations and to communicate system specifications and construction blueprints. Recent advances in on-line collaboration and modeling tools have raised the expectations for quickly making changes to models, resulting in frequent changes to layouts, text, and/or symbols in schematic drawings. Automating the creation of high-quality schematic views from geographical and topological models can help engineers produce and update drawings efficiently.

This paper introduces four methods for automatically producing *schematic railway drawings* with increasing level of quality and control over the result. The final method, implemented in the open-source tool that we have developed, can use any combination of the following optimization criteria, which can have different priorities in different use cases: width and height of the drawing, the diagonal line lengths, and the number of bends.

We show how to encode schematic railway drawings as an optimization problem over Boolean and numerical domains, using combinations of unary number encoding, lazy difference constraints, and numerical optimization into an incremental SAT formulation. We compare drawings resulting from each of the four methods, applied to models of real-world engineering projects and existing railway infrastructure. We also show how to add symbols and labels to the track plan, which is important for the usefulness of the final outputs. Since the proposed tool is customizable and efficiently produces high-quality drawings from railML 2.x models, it can be used (as it is or extended) both as an integrated module in an industrial design tool like RailCOMPLETE, or by researchers for visualization purposes.

1. Introduction

Engineering schematics of railway track layouts are used for several purposes: serving as construction blueprints, visualizations on train dispatch workstations, infrastructure models in timetabling software, specifications for interlocking control systems, and more. Because of the large distances involved, geographically accurate drawings are not always suitable for communicating an overview that can help with analyzing and

Correspondence and offprint requests to: Christian Johansen, Norwegian University of Science and Technology, Gjøvik, Norway
e-mail: christian.johansen@ntnu.no

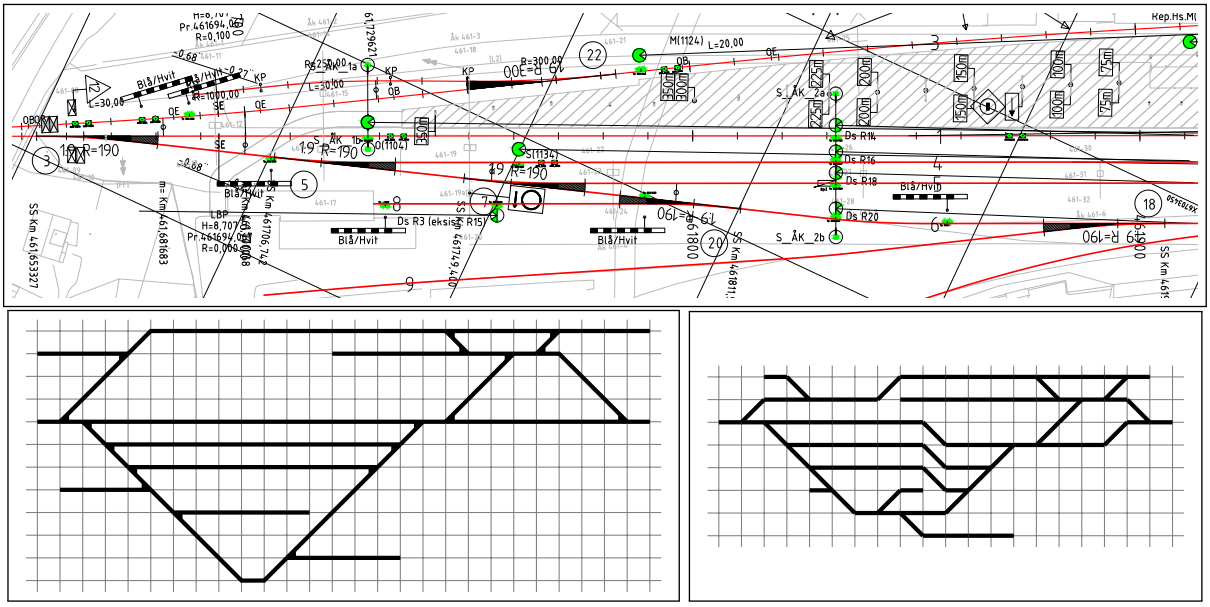


Fig. 1. Example cut-out from a geographical railway drawing (top) and two corresponding full-station schematic layouts, optimized for bends (bottom left) and optimized for height/width (bottom right). See on page 20 our tool's optimization options.

reasoning about the railway models. Instead, many disciplines use *schematic* representations of infrastructures to provide a compressed overview, e.g., shortening sections of the railway that have low information density. Figure 1 compares a geographically correct drawing against two alternative schematic renderings of the same model for two different purposes: the left-hand drawing is suitable for a schematic engineering drawing while the right-hand drawing is more suitable for a dispatcher workstation visualization where a highly compressed view is better. Producing schematic drawings like these involves practical and aesthetic trade-offs between intended structure, simplicity, and geographical accuracy.

Perhaps the most well-known railway schematics are the metro maps for passengers, popularized by the iconic Tube Map of the London Underground. When designing metro maps, removing and compressing geographical information better conveys topological structure (e.g., useful for finding transfers) and sequential information along lines (e.g., for finding your stop).

Methods for automatically producing metro maps have been surveyed in [25]. The main approaches are iterative and force-directed algorithms for gradually transforming a geographical network map into a simpler presentation [2, 8], and mixed integer programming methods for finding exactly grid-structured and rigidly optimized solutions [16, 18]. For railway drawings the convention is to use only horizontal, vertical, and diagonal lines (at 45°). The problem of drawing graphs optimized for size and/or bends using only horizontal and vertical lines (so-called orthogonal drawings) can be solved by efficient algorithms [23], but adding diagonal lines in general makes the problem NP-complete [16, 17].

Schematic railway drawings used for engineering are usually more strictly constrained than metro maps, but still have large variety in different versions produced for different engineering use cases, project stages, and operational scenarios. Especially in construction projects for new railway lines or upgrades, frequent changes are made in coordinated 2D, 3D, geographical, and schematic models of the railway infrastructure. This can cause much repeated manual work in updating and cross-checking these models after every change in the construction design work in several engineering and construction categories, such as tracks, signaling and interlocking, catenary, cables, telephony.

Automatically producing consistent and high-quality schematics from other models has great potential to increase the efficiency and quality of the documentation, speed up cross-discipline communication during design and construction phases, and also opens up for easier data transfer to other tools. For example, an engineer working on a geographical CAD model may be hindered in performing capacity analysis because

importing a network model into a capacity tool may require also inputting a track plan for simulation overview (see, e.g., [13]).

In this paper we develop methods for producing a type of schematic track plan which is suitable for infrastructure within a single corridor, meaning that each point on each track can be located on a common linear axis. We call this a *linear schematic drawing* (see Definition 1 on page 6). This is a common drawing used for many purposes in construction projects, where drawings typically show placement of tracks and track-side equipment on a single station or along a single corridor. More generally, this problem concerns network structures that are oriented along a linear axis, such as highways, railways, or public transit systems, but may also be extended to encompass routing in electronic design (see e.g. the problem description for VLSI routing in [19]). On larger scales with multiple corridors, the visualization may be split into individual corridors, as in our setting, but for some applications, such as an overview of a national railway network or a city metro network, the single corridor assumption will not work well, and other approaches (see e.g. [16, 18]) may be more relevant.

Linear schematic drawings specifically have little coverage in the literature. A specialized algorithm presented in [7] computes corridor-style drawings, but does not guarantee that switch shapes are preserved, and does not offer choices in optimization criteria. For comparison, we apply our method to examples taken from [7], among others; see Figure 14 on page 19. Another algorithmic approach described in [22] has similar goals, but does not automatically produce high-quality results and relies on interactive guidance from the user and manual post-processing.

Graph drawing techniques (see [9, 10] for a general overview) have been developed for a great number of different use cases. Most closely related to engineering drawings are the orthogonal layout methods (see e.g. [20, 23]) and storyline visualizations (see e.g. [24]). However, most approaches from the graph drawing literature, including orthogonal layout methods, produce outputs that have a distinct style and are not suited to be customized to adhere to engineering drawing conventions.

Instead, we have solved the problem by modeling engineering drawings as mathematical optimization problems using constraints formulated as Boolean satisfiability and difference constraints. We present how different available constraint programming systems can be used to express our constraints, solve optimization problems, and produce high-quality engineering drawings.

The main contributions of this paper are:

1. We formally define and describe the problem of *linear schematic railway drawings* in Section 2.
2. We introduce four mathematical models for schematic plans, and compare their strengths and weaknesses in Section 3.
3. We develop an open-source tool that can be used by railway engineers to visualize railway infrastructure, and demonstrate its performance and output on real-world infrastructure models in Section 4.

Our tool[1] can be used as a module, e.g., integrated in the RailCOMPLETE engineering framework; but it can also be used as a standalone tool by researchers and developers working on new techniques for analysis and verification, e.g. on interlockings or capacity and timetabling, who can greatly benefit from low-effort, high-quality visualizations in order to improve communication, usability, and for lowering the barrier for adoption of their tools and techniques. Our tool takes input railML files, which are widely available among railway engineers as it is a standard description format for railway infrastructure. The tool also has options for placing symbols besides a track in the schematics.

This paper is an extended version of the conference paper [14] containing three new sections: Section 2.4 describing in detail the problem of drawing symbols around the tracks in a schematics; Section 3.3 introducing a fourth method of drawing in SAT using a grid-based encoding; and Section 4 which is a considerable enlargement of the two paragraphs that appeared in the conference proceedings. Besides, a few corrections were made and more care given to the presentation and exemplification of our formalizations, often providing more details.

[1] See the `railplot` web page: `https://github.com/luteberget/railplot`

2. Problem definition and formalization

In this section we define and formalize the problem of drawing railway infrastructure. First, we define the input data model that is used to represent the relevant information about railway infrastructure for creating schematic drawings (Section 2.1 and Section 2.2). Then, the problem definition is presented in two parts:

1. The linear schematic track drawing (Section 2.3) defines the drawing of lines representing the topology of railway tracks on the infrastructure, and forms the backbone for the rest of the drawing.
2. The linear schematic symbol placement (Section 2.4) assumes that the track drawing problem is solved, and defines a restricted model for placement of relevant symbols and labels.

The definitions and assumptions presented below were designed by the authors working together with software developers and railway engineers with the purpose of formalizing and automating complex tasks in the context of railway construction engineering. We were particularly inspired by the requirements of the *signaling* sub-discipline, being also one that we are quite familiar with. The trade-offs made in these definitions, and in modeling and solving the problem in the following sections, are intended to balance:

1. *The needs for low complexity in small-scale software development:* this concerns software suppliers such as Railcomplete AS who are developing software to be used in railway design stages (e.g., for signaling designs) which have to graphically present railway infrastructure schematics both on the screen, to the railway engineer using the software, but also printed or exported for further design stages. Using an off-the-shelf mathematical solver allows developers to focus on user-friendly tooling instead of implementing and maintaining a complicated algorithm.
2. *Short running times in interactive software:* this concerns railway engineers using the software. Engineers are often willing to sacrifice perfect optimality with respect to the given criteria if the calculations run orders of magnitude faster.
3. *The quality requirements for actually using the results as engineering artifacts:* this concerns railway engineers producing schematic drawings and also their engineering organization managing its quality assurance policy. For some use cases, such as engineering drawings, formalization, consistency, optimality, and reproducibility may be key concerns. For other use cases, such as visualizations used in software tools, approximate optimality and by-hand adjustments may be more valuable. The methods presented below have different strengths and weaknesses in these aspects.

2.1. Linear positioning system

It is a common practice in railway engineering to use a linear reference positioning system (in British English called mileage, in German Kilometrierung, and in railML absolute position), which assigns a scalar value to each point on, or beside, a railway track. The value corresponds approximately to the traveling distance along a railway corridor from a reference point (which is often a known point on the central station of the network). For a single track, the linear reference system may simply be the arc length from the start of the track's center-line curve. Double tracks and side tracks are typically mapped to the linear reference position by geometrically projecting each point onto a reference curve. The projection's target curve may either be a selected *reference track* (see Figure 2), or another curve that does not necessarily coincide with a track, such as the geometrical center-line of the corridor. For the rest of this paper, we assume that all locations are already given in such a linear reference system. For network-oriented models obtained from analysis software, such as simulators and time-table planners, and exchange formats such as railML, the linear positions are the main position attribute given to objects, and should always be available. For models obtained from CAD software, the linear positions must be computed by geometrical calculations by using the track geometry, and this is a basic feature of all railway-related CAD extension softwares.

2.2. Track network representation

Different track segments are connected together at *switches* in a graph-like network. The mathematical definition of a graph is too abstract for many engineering use cases. Some applications use a *double node graph* [12], or describe tracks as nodes with two distinct sides [1]. For a schematic plan, we model switches

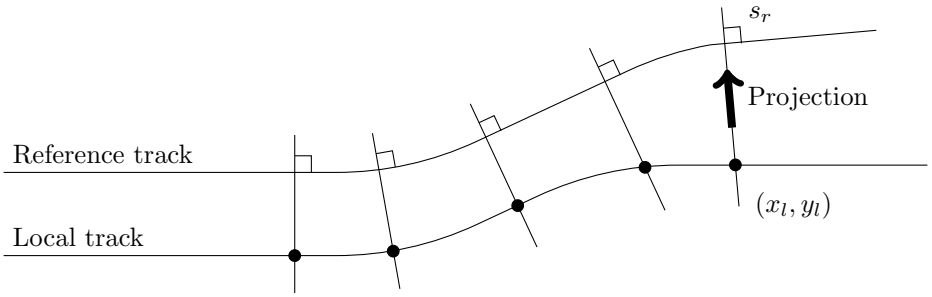


Fig. 2. Linear reference position calculated by projection onto a reference track.

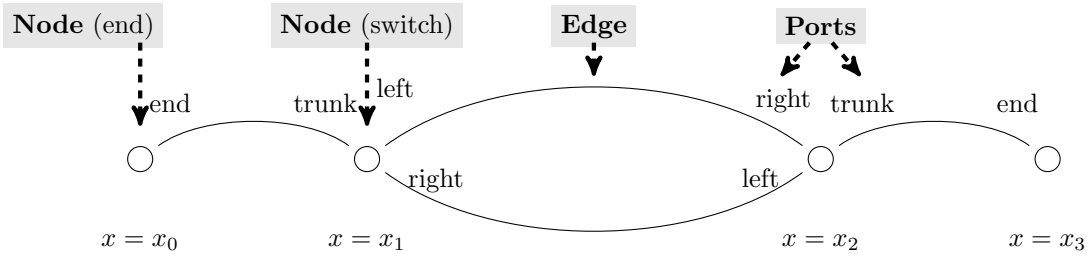


Fig. 3. Graph representation of linearized track plan. Nodes are ordered by an x coordinate, and have a given type which determines which ports it has, e.g., a switch node has *trunk*, *left*, and *right* ports. Edges connect ports on distinct nodes.

and crossings as graph nodes which have a given set of *ports* (Figure 3 presents all our modeling elements). Each end of each edge connects to a specific port on a specific node. Model boundaries and track ends are also represented as nodes with a single port.

Each location where tracks start/end or intersect with other tracks is represented as a node of a given class. The classes used in this paper are ends, switches, crossings, and flyovers (shown in Figure 4 with all their representative variants). Each class comes with a different set of drawing requirements. For example,

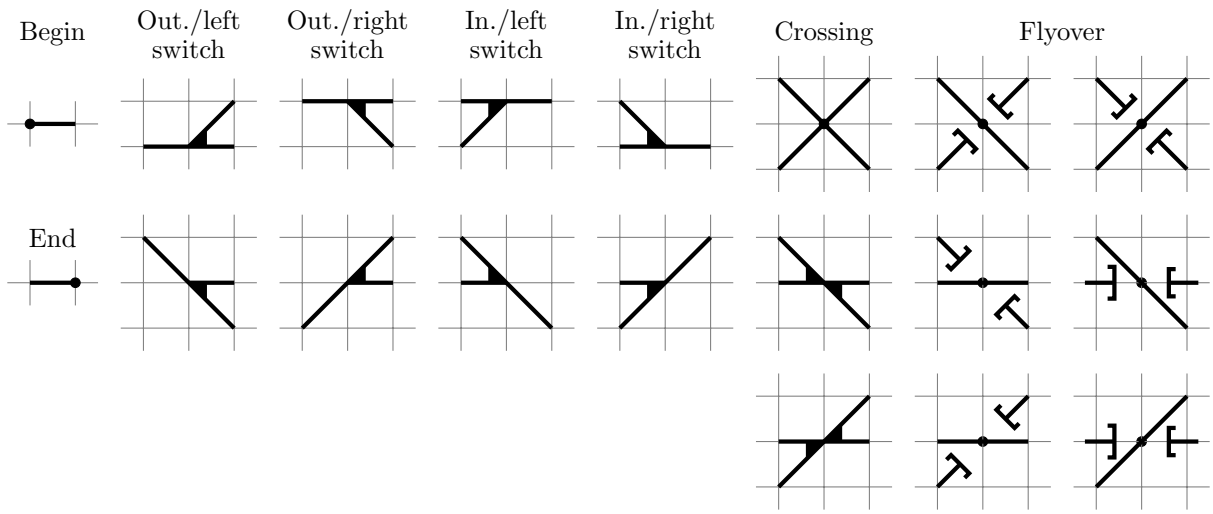


Fig. 4. Node classes and their drawing variants. Begin/end nodes have one variant each. Switches are divided into four classes (each with two variants) based on their orientation (incoming or outgoing) and their course (deviating left or right). Crossings have three variants, and flyovers have six variants.

a switch is oriented such that its branching edges (left/right) point either up (called an outgoing switch) or down (called an incoming switch), seen in the positive direction of the linear positioning system, and each switch class can be drawn in two different variants, chosen freely, one with the trunk and straight leg directed horizontally and another with the deviating leg directed horizontally.

2.3. Linear schematic track drawing

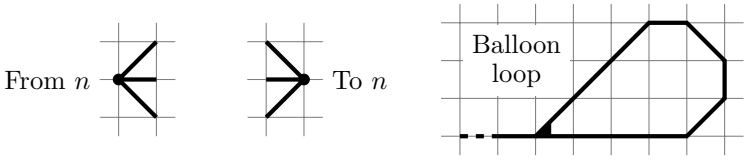
A linear schematic drawing algorithm is a core concept in our formalization.

Definition 1. A linear schematic track drawing algorithm $d : (N, E) \to L$ assigns a set of line segments L to each edge in the set E of edges connecting the set of nodes N, where:

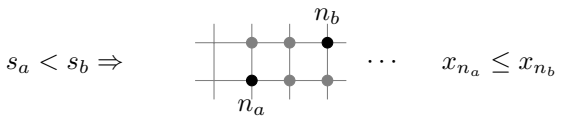
- $N = \{n_i = (c_i, s_i)\}$, where $c_i \in C$ is a node class, and $s_i \in \mathbb{R}$ is a linear position distinct from other nodes' positions.
- $E = \{e_j = (n_a, p_a, n_b, p_b)\}$, where $n_a, n_b \in N$ are two nodes where $s_a < s_b$ and p_a, p_b are distinct, available ports on the referenced nodes.
- $L = \{(e_j, l_j)\}$, where l_j is a polyline, representing the drawing of edge $e_j \in E$, and defined by a sequence of points in $\mathbb{R}^2$, $\langle (x_1^j, y_1^j), (x_2^j, y_2^j), \ldots, (x_n^j, y_n^j) \rangle$. The polyline consists of the line segments connecting consecutive points in this sequence.

The definition of a track drawing algorithm in itself does not ensure that the output drawing is suitable for reading. To ensure a usable output we establish a set of *criteria for drawing quality* based on engineering conventions and aesthetic judgements. We divide the criteria into *hard constraints*, that all drawings must satisfy and that we can base mathematical models on, and *soft constraints*, which are optimization criteria that can be prioritized differently in different use cases. We base our models on the following hard constraints provided by railway signaling engineers (from Railcomplete AS):

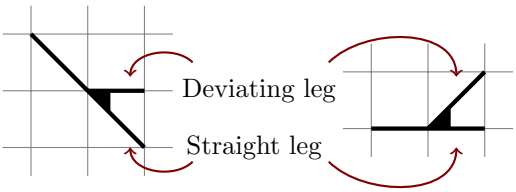
(A) **Octilinearity**: the lines representing tracks should be either horizontal, or diagonal at 45°. This property contributes to the neat look of a schematic drawing, while also giving a visual clue that the drawing is not fully geometrically accurate. When loops are present in the infrastructure, vertical lines may also be allowed, such as in the *balloon loop* used on many tram lines.



(B) **Linear order**: the reference mileages of locations on the infrastructure should be ordered left-to-right on the schematic drawing to give a clear sense of sequence, which is useful when navigating the infrastructure and reasoning about train movements.



(C) **Node shapes**: switches split the track on the *trunk* side into a left and a right leg on the *branch* side. Left and right should be preserved so that the layout can be traced back to the geography. Since one of the legs of the switch is typically straight and the other is curved, it is also desirable to preserve the straight leg's direction relative to the trunk.



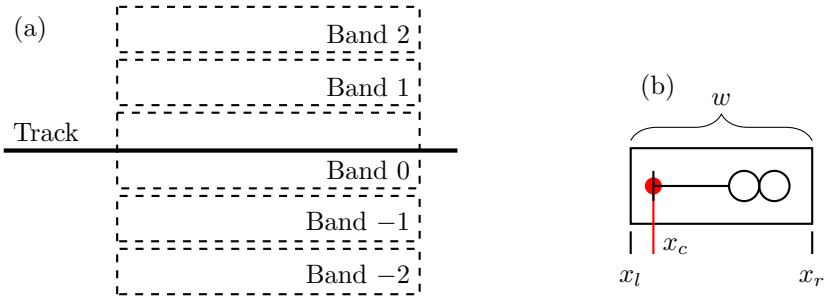
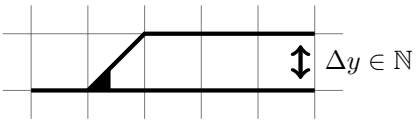


Fig. 5. (a) A track, drawn as the black horizontal line in the middle, defines a set of *bands* of fixed width around it. (b) A symbol is defined by its width w and the relative horizontal position of the location origin s, and is placed in the drawing by choosing a horizontal center coordinate x_c from which the left and right edge coordinates x_l and x_r can be computed.

(D) **Uniform horizontal spacing**: parallel tracks are typically required to be drawn at a specific distance from each other, which we normalize and say that y coordinates take integer values. Note that x coordinates have no such restriction, but consecutive nodes will often be placed at integer-valued distances to fulfill the octilinearity constraint.



Even with the above constraints fulfilled, there is no guarantee that the drawing output of an algorithm can be deemed of high-quality. For this we use the following soft constraints as optimization criteria:

(i) **Width** and **height** of the drawing.
(ii) **Diagonal line length**, the sum of length of non-horizontal line segments.
(iii) **Number of bends**, i.e. the number of direction changes on lines.

These criteria have different priorities in different use cases. For example, a signaling schematic might be optimized to have a minimum amount of diagonal lines to neatly show several concurrent train movements and their relative progress, while a dispatch control station schematic might be optimized for width to fit more infrastructure into a computer screen.

Several or all of the criteria can be combined into an optimization objective, either by a scoring function, or more commonly, by simply ordering the objectives and performing lexicographical optimization on each objective in turn. Our tool (detailed in Section 4) provides options for *ranking the objectives*.

2.4. Linear schematic symbol placement

A railway engineering schematic often features a large amount of different symbols and labels (recall the example in Figure 1). In a railway signaling schematic, signals and related equipment are drawn on the schematic, and the diagram is richly annotated with traveling distances, area limits, comments, and more.

In some cases, the symbols and labels can be placed onto a well laid-out track plan without needing to change the track plan to accommodate other symbols and labels, but in other cases the track layout must be drawn in a way that accounts for the amounts and sizes of symbols and labels.

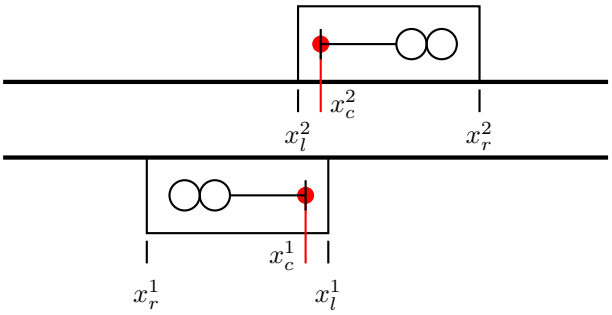
The general label placement problem is known in the graph-drawing literature. We focus here on a simplified formalization of the problem which is useful for several variants of railway infrastructure schematics, where each of the lines representing a track has a specified number of *bands* around it, and where symbols may be placed by specifying a single linear position, cf. Figure 5.

We assume that the symbols themselves have already been assigned to one of the bands and each has a fixed symbol size. This assumption is inspired by schematic *signaling* drawings, where objects such as signals, balises, point machines, derailers, etc., are drawn at their specified locations around the track.

The symbols may then be placed subject to the following hard constraints (on top of the ones from Section 2.3):

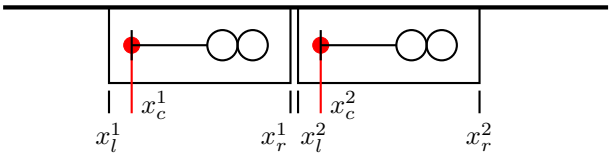
(E) **Linear symbol order**: symbols across all tracks and bands should be displayed according to the linear reference system of Section 2.1 by ordering them based on their location origin, i.e., their center coordinates x_c. In other words, for any two consecutive symbols to be well drawn their center points x_c^1, x_c^2 should be ordered as in the following:

$x_c^1 \leq x_c^2$:



(F) **Non-overlapping symbols**: for the symbols that belong to the same band on the same track, an additional ordering constraint applies to ensure that the symbols are not drawn overlapping. Thus, for two symbols within a band that are ordered increasingly on the linear reference system, the horizontal coordinate of the right end of the first symbol x_r^1 must be less than the left end of the second symbol x_l^2:

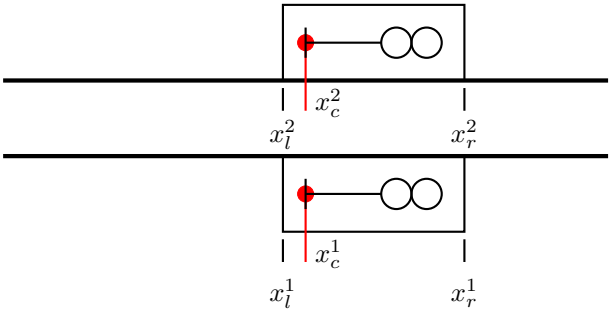
$x_r^1 \leq x_l^2$:



This also means that symbols belonging to the same band on the same track are not allowed to have the alignment constraint described below.

(G) **Alignment constraints**: in addition to the linear symbol order, some symbols may be placed symmetrically on several parallel tracks, and have an additional specified equality constraint, i.e., for each pair of symbols in the set of symbols associated with an alignment constraint, the center points, x_c^1 and x_c^2, must be equal:

$x_c^1 = x_c^2$:



We add an alignment constraint when two or more symbols on different bands have linear positions that differ by less than 1m.

In addition to these constraints, it is also desirable to keep the symbols proportionally distributed relative to the linear position system on the horizontal axis, so that the drawing gives a more accurate impression of the location of objects. We use the following optimization criterion:

- **Proportional placement**: assume a track stretching from linear position s_1 to s_2 has been assigned

horizontal coordinates x_1 and x_2. A symbol at linear position s_s has the *proportional* placement x_s^{prop}:

$$x_s^{\text{prop}} = x_1 + (x_2 - x_1)\,\frac{s_s - s_1}{s_2 - s_1}$$

The proportional placement optimization criterion is to minimize:

$$\sum_{s \in S} |x_s^{\text{prop}} - x_s|$$

Note that the constraints on symbol placement may not be satisfiable if the track plan is decided in advance and there is not enough room on the drawing for all the symbols. This means that, in general, the track plan problem and the symbol placement problems are interacting, and may benefit from being solved simultaneously.

Actually, this reflects one of the key motivations for automating schematic drawings from the railway engineer's perspective: when adding or modifying symbols on the schematic track drawing, some additions or modifications require reshaping the backbone of the drawing, represented by the track lines. If this happens at a late stage of the project, there may not be sufficient time and resources available to update the schematic and lower-quality drawings will be used instead, as a shortcut.

Instead of full simultaneous optimization of both the track plan and the symbol placements, we also define the following simplified, heuristic approach:

Definition 2. The *weakly interacting* simultaneous track and symbol drawing problem is defined as follows: whenever solving the symbol drawing problem fails because there is not enough room on a track segment, re-solve the track drawing problem with an additional constraint that increases the length of that track.

3. Model definitions and drawing algorithms

This section describes the four different models of linear schematic drawings that we have developed. All models use a pre-processing step which orders edges vertically, described in Sec. 3.1. The first method is a linear programming formulation (Section 3.2) where edges can have up to two bends, and the middle section of each edge is a horizontal line at a certain *level*, i.e. y coordinate. The resulting optimization problem is efficiently solvable, but has some drawbacks in visual quality. The second method introduces Boolean choice variables to mitigate the shortcomings of the linear programming formulation, and use instead a SAT solver to solve a direct representation of choices for each point on the whole drawing grid (Section 3.3). The third model is a different Boolean model where the choices for where different edges can appear is partially implicit in the variable domains, which we call the cross-section SAT encoding (Section 3.4). Finally, we re-visit the levels-based formulation in combination with Boolean choice (Section 3.5), using lazy solving of difference constraints to optimize the Boolean/numerical model (keeping the maximum of two bends per edge).

An approach for adding symbols and labels to engineering drawings is presented in Section 3.6. This approach is developed separately as an optimization problem that can be combined with the rest of the drawing methods that we present in this section.

Section 3.7 summarizes the quality and performance trade-offs between the four methods. Comparison figures 8, 11, and 14 demonstrate the strengths and weaknesses of each approach, while Table 2 and Table 3 describe their relative performance and quality. We use several well-known SAT encoding techniques without explaining them in detail, such as finite set encodings, unary number encodings, at-most-one, and optimization using binary search. These techniques are described in [4, 5].

The presentation of these four methods is both important for research purposes to show also weaknesses of alternative simpler approaches (instead of only showing the most powerful of the methods), but is also intended to be educational, where the earlier simpler methods introduce only some of the concepts (used also in later methods), making the whole presentation easier to follow, more organized and ready for comparisons.

3.1. Vertical ordering relation on edges

From the nodes and edges defined as inputs to the linear schematic drawing algorithm, it is possible to derive a vertical ordering relation $<_E$ on the set of edges. This relation is a strict partial order relating edges whose

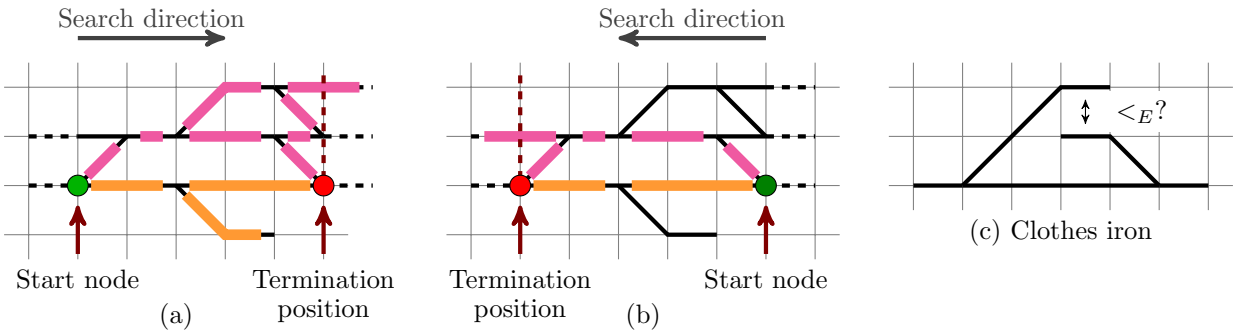


Fig. 6. A search procedure starting in each node produces a set of tuples for the edge vertical order relation $<_E$. Figures (a) and (b) show two different start nodes and search directions, where the lighter, orange edges are all below darker, magenta edges. Figure (c) shows an input on which the procedure cannot decide an ordering.

linear position intervals intersect, i.e., it relates each pair of edges e_a from n_{a_l} to n_{a_r}, and e_b from n_{b_l} to n_{b_r}, where:

$$]s_{a_l}, s_{a_r}[\ \cap \ ]s_{b_l}, s_{b_r}[\ \neq \emptyset.$$

Such a relation can be established by considering paths starting in each of the branch-side ports of each switch, crossing, and flyover (cf. Figure 4). For example, an outgoing switch with branch-side edges e_a and e_b connecting to its right and left ports, respectively, will obviously have $e_a <_E e_b$. Each edge connected to the outgoing edges from the other side of e_a and e_b will also be ordered vertically, and so on until either of the following termination conditions are fulfilled:

(C1) The two sets of edges meet in another node.
(C2) One of the sides has no more edges to follow.

More precisely, we define $<_E$ by the following. Let $G = (N, E)$ be the graph from Definition 1. We first look in the positive direction on the linear reference axis. We define a vertical order relation $<^i_E$ for each node $n_i \in N$. If n_i has less than two ports on the side of increasing linear position, $<^i_E$ is empty. However, if the node has two ports on the side of increasing linear position, let the edges connected to these ports be e_l, the lower edge, and e_h, the higher edge. For example, in an outgoing switch node (cf. Figure 4), these correspond to the right and left ports, respectively.

For any node n_j with $s_i < s_j$, define the directed graph $H_{]i,j[}$ containing:

- The subset of nodes from G with positions in the open interval $]s_i, s_j[$, along with any number of fresh nodes (i.e. the nodes n_i and n_j are not included).
- The subset of edges from G which have at least one end connected to a node from the open interval $]s_i, s_j[$, directed in the direction of increasing linear position. If an edge connects to a node from G which is not included in $H_{]i,j[}$, that connection is replaced with a connection to a distinct fresh node.

We are looking for those nodes n_j such that, in $H_{]i,j[}$, the set of reachable edges when starting from e_l are disjoint from the set of reachable edges when starting from e_h (termination condition (C1)), see Figure 6(a). Also, the linear position interval of each edge reachable from e_l should have a non-empty intersection with at least one edge reachable from e_h, and vice versa (termination condition (C2)). The node n_j which has the highest position s_j while still fulfilling the above criteria, is called the *termination position*.

Each edge e_x reachable from e_l in $H_{]i,j[}$ is below all edges e_y reachable from e_h in $H_{]i,j[}$ whenever this pair of edges has intersecting linear position intervals, in which case we have $e_x <^i_E e_y$.

For the direction of decreasing linear position we apply the same argument with horizontal directions reversed (see Figure 6(b)). Finally, the relation $<_E$ is defined as the union of the relations from each node,

$$<_E = \bigcup_{n_i \in N} <^i_E .$$

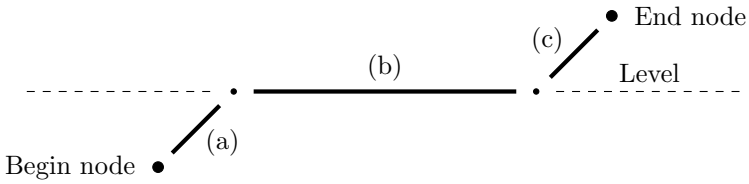


Fig. 7. The *edge level model* divides the edge into three sections on the horizontal axis: (a) the initial diagonal section from the left-most node to the edge level, (b) the middle horizontal section connecting the two diagonal sections, (c) the final diagonal section reaching the right-most node from the edge level. Any of these may have zero length.

Remark 3. Unconnected graph components must still be explicitly ordered, and the same for some connected topologies such as the clothes iron example in Figure 6(c). These are usually easy to decide from, e.g., a geographical model, and this situation occurs rarely, in our experience.

3.2. Level-based linear programming encoding

We start by giving a constraint system on linear equations over continuous numerical variables which fulfills the hard requirements from Section 2.3 and can be solved efficiently by linear programming (we used the CBC solver v2.9[2]). Later, the shortcomings of this model will motivate the introduction of Boolean and integer-valued variables and a SAT problem formulation.

The main idea for this first model is based on the observation that (for many drawings) edges are mostly horizontal, and they can thus be modeled as occupying a certain y-coordinate (the "level") over a horizontal interval. A complete drawing can then be produced by connecting the ends of the horizontal segments by shorter diagonal segments. This *level* concept is more of a mental model that railway engineers are using when producing drawings by hand (we are not aware of such practices being described in publicly available literature). We formalize this level-based style of schematic drawing here to take advantage of it for producing drawings automatically.

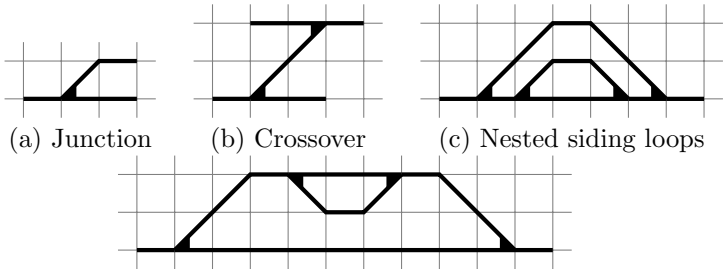
For each node n_i we use two real variables, x_i and y_i, representing the schematic coordinates of nodes. For each edge e_i we use one real variable l_i representing the edge's *level*. This builds in an assumption that each edge is drawn in three parts as explained in Figure 7. We introduce the following constraints:

1. Node location ordering for successive nodes n_i, n_j gives $x_i \leq x_j$, corresponding to the linear order requirement (from Sec. 2.3(B)).
2. Node location distance for nodes n_i, n_j connected by an edge e_k, where $s_i < s_j$, gives $x_i + |l_k - y_i| + |y_j - l_k| + q_k \leq x_j$, where q_k is 0 if the edge connects an outgoing switch to an incoming switch with the same branching direction, and 1 otherwise. This creates room for a horizontal line segment if needed. The sign of the absolute value terms is determined statically (not part of the linear programming) by the node class and variant. This constraint corresponds to the octilinearity requirement (from Sec. 2.3(A)).
3. Edge level ordering for edges: $e_i <_E e_j$ gives $l_i + 1 \leq l_j$, corresponding to the node shape requirement (from Sec. 2.3(C)).
4. Edge levels are related by switches, i.e.: each switch node n_i constrains the trunk-side edge e_j and the straight branch-side edge e_k to be at the same level as the node ($y_i = l_j = l_k$) corresponding to the node shape requirement.

Note that the uniform horizontal spacing constraint (from Sec. 2.3(D)) is implicit in these equations. Now we have the following criteria available for optimization:

- **Width of the drawing.** Take the node n_i with the lowest s_i, and the node n_j with the highest s_j. Then the width of the drawings is $x_j - x_i$.

[2] Part of the COIN-OR project 2018: `https://projects.coin-or.org/Cbc`



(a) Junction (b) Crossover (c) Nested siding loops

(d) Ladder sidings

Fig. 8. Output examples for the linear programming method. The junction (a) and nested sidings (c) are correctly drawn. The crossover (b) uses 2 units for the diagonal, where 1 would be sufficient, because each edge requires a *level* distinct from other edges with intersecting linear position intervals. The ladder sidings (d) are unnecessarily wide because node shape variants are not included (compare with Figure 11(b)).

- **Height of the drawing.** The height of the drawing is not directly expressible in this model, but can be approximated by summing the vertical level difference of edges. For pairs of edges e_i, e_j where $e_i <_E e_j$, the vertical level difference distance is $l_j - l_i$.

Some output examples from the linear programming solution are shown in Figure 8. Although efficiently solvable, this linear programming solution has a main drawback in that it is not able to choose between different alternatives for drawing a node. For example, in the so-called ladder configuration shown in Figure 8(d), much space is wasted on diagonal lines going to the top-most level, when the two topmost switches could have been rotated to produce a simpler drawing. Also, each edge needs to have a y value distinct from other edges with intersecting x intervals, even if it is drawn only with diagonals, such as in Figure 8(b), which contributes to inefficient use of space. Both these shortcomings will be improved by the level-based Boolean formulation in Section 3.5.

3.3. Direct grid-based SAT encoding

The level-based representations do not represent the shapes of edges explicitly at each coordinate, and thus cannot insert bends at arbitrary locations, something which is needed to pack drawings together more tightly. This section presents a straight-forward grid-based SAT encoding of the schematic plan where each point on a grid is associated with a choice of any node, and each cell with a choice of edge shape.

First, assume that we know a grid size $w \times h$ which is sufficiently large to contain a schematic drawing of a given infrastructure. Each grid cell may be either blank, be occupied by a node $n \in N$, or be point on an edge $e \in E$. Furthermore, each horizontal line $l^{\text{Straight}}_{x,y}$, diagonal upwards line $l^{\text{Up}}_{x,y}$, and diagonal downwards line $l^{\text{Down}}_{x,y}$ starting at each point on the grid is either drawn or not. These are the variables in the problem:

$$p_{x,y} \in \{\text{None}\} \cup N \cup E, \qquad l^{\text{Up}}_{x,y}, l^{\text{Straight}}_{x,y}, l^{\text{Down}}_{x,y} \in \mathbb{B}, \qquad \text{for } (x,y) \in [1,w] \times [1,h]$$

We use the standard one-hot encoding for encoding the selection of an element from a finite set into the Boolean satisfiability problem (see [4] for an overview of the standard techniques). See Figure 9 for an overview of the variable definitions through an example.

The following constraints are used:

- Grid points adjacent to active edges cannot be unused. For lines starting in (x, y):

  $$l^{\text{Up}}_{x,y} \Rightarrow (p_{x,y} \neq \text{None} \wedge p_{x+1,y+1} \neq \text{None}),$$

  $$l^{\text{Straight}}_{x,y} \Rightarrow (p_{x,y} \neq \text{None} \wedge p_{x+1,y} \neq \text{None}),$$

  $$l^{\text{Down}}_{x,y} \Rightarrow (p_{x,y} \neq \text{None} \wedge p_{x+1,y-1} \neq \text{None}),$$
- Only one of the two possible diagonals crossing a grid square may be used:

  $$\neg l^{\text{Down}}_{x,y} \vee \neg l^{\text{Up}}_{x,y-1}$$

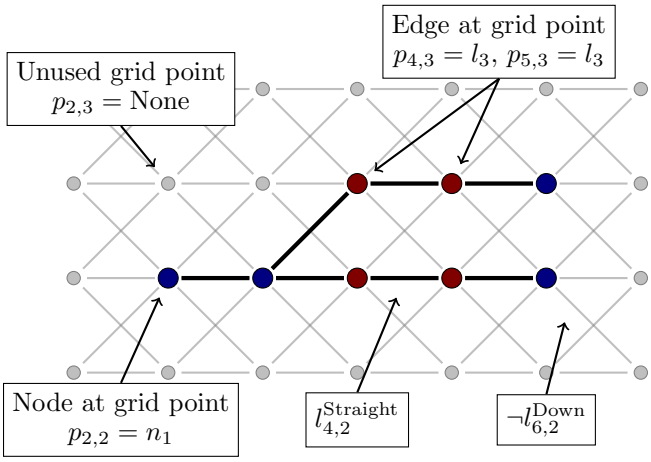


Fig. 9. Direct SAT representation. Each grid point is either gray (unused), blue (occupied by a node), or red (occupied by an edge). Each line segment is either gray (deactivated) or black (activated).

Variant	$l^{\text{Down}}_{x-1,y+1}$	$l^{\text{Straight}}_{x-1,y}$	$l^{\text{Up}}_{x-1,y-1}$	$l^{\text{Down}}_{x,y}$	$l^{\text{Straight}}_{x,y}$	$l^{\text{Up}}_{x,y}$
$a_1(x,y)$	true	false	false	true	false	false
$a_2(x,y)$	false	true	false	false	true	false
$a_3(x,y)$	false	false	true	false	false	true
$a_4(x,y)$	true	false	false	false	true	false
$a_5(x,y)$	false	true	false	false	true	false
$a_6(x,y)$	false	true	false	false	false	true
$a_7(x,y)$	false	false	true	false	true	false

Table 1. At each grid point (x, y) that is occupied by an edge $p_{x,y} = e$, $e \in E$, there are seven valid combinations of active adjacent edges, $a_1, \ldots, a_7$, where each combination is a conjunction of the variables named in the column headers. The value *true* in the table indicates inserting the variable positively, while the value *false* indicates that the variable is negated in the formula.

- Each node is at exactly one grid point in the drawing:

  $$\forall n \in N : \text{exactlyOne}(\{p_{x,y} = n \mid (x,y) \in [1,w] \times [1,h]\}),$$

  where *exactlyOne* is encoded as one clause and one *atMostOne* constraint using a standard hierarchical encoding.
- At each node $n \in N$ at grid point $p_{x,y} = n$, the active edges at that point must correspond to an allowable set of adjacent line segments. For example, for a left outgoing switch n_1 there are two variants v_1 (straight) and v_2 (slanted):

  $$\forall (x,y) \in [2, w-1] \times [2, h-1] : (p_{x,y} = n_1) \Rightarrow v_1(x,y) \vee v_2(x,y),$$

  $$v_1(x,y) = \neg l^{\text{Down}}_{x-1,y+1} \wedge l^{\text{Straight}}_{x-1,y} \wedge \neg l^{\text{Up}}_{x-1,y-1} \wedge \neg l^{\text{Down}}_{x,y} \wedge l^{\text{Straight}}_{x,y} \wedge l^{\text{Up}}_{x,y},$$

  $$v_2(x,y) = l^{\text{Down}}_{x-1,y+1} \wedge \neg l^{\text{Straight}}_{x-1,y} \wedge \neg l^{\text{Up}}_{x-1,y-1} \wedge l^{\text{Down}}_{x,y} \wedge l^{\text{Straight}}_{x,y} \wedge \neg l^{\text{Up}}_{x,y}.$$

  Note that the choice of node class variant is not explicitly represented as a variable, but is implicit in the mutually exclusive choices of enabled line segments neighboring the grid point.
- Also at each node, the nearby line segments must lead to a node with the correct edge. For example, say that $v_1(x,y)$ above connects nodes n_1 and n_2 through edge e_1 on its *left* port. Then we have:

  $$v_1(x,y) \Rightarrow (p_{x,y+1} = e_1 \vee p_{x,y+1} = n_2)$$
- At each edge $e \in E$ at grid point $p_{x,y} = e$, exactly two neighboring edges must be active, and these cannot be diagonals going in different directions.

So for conjunctions $a_1(x,y), a_2(x,y), \ldots$ corresponding to each line of Table 1, we get:

$$\forall e \in E : \forall (x,y) \in [1,w] \times [1,h] : a_1(x,y) \vee a_2(x,y) \vee \ldots$$

- Also at each edge, the nearby line segments must lead to the correct node. For example for an edge e_1 connecting nodes n_1 and n_2:

  $$\forall (x,y) \in [1,w] \times [1,h] :$$

  $$\left(p_{x,y} = e_1 \wedge e^{\text{Straight}}_{x-1,y}\right) \Rightarrow (p_{x-1,y} = e_1 \vee p_{x-1,y} = n_1)\,,$$

  $$\left(p_{x,y} = e_1 \wedge e^{\text{Straight}}_{x,y}\right) \Rightarrow (p_{x+1,y} = e_1 \vee p_{x+1,y} = n_2)\,,$$

  and similarly for diagonal (up/down) lines.
- The linear ordering is imposed as follows for two consecutive nodes n_1 and n_2:

  $$\forall (x_1,y_1) \in [1,w] \times [1,h] : (p_{x_1,y_2} = n_1) \Rightarrow (\forall (x_2,y_2) \in [1,x_1-1] \times [1,h] : \neg(p_{x_2,y_2} = n_2))$$

The grid encoding in principle requires a bound on the width and height of the drawing. To avoid specifying this bound, we can start by solving the problem with a lower bound estimate on the drawing's size, and, whenever the problem is shown to be unsatisfiable by the solver, increase the size of the drawing. The increased-size problem can benefit from incremental SAT solver calls, as the structure of the smaller-size drawing is preserved exactly as it was. When increasing the bound, one needs also to choose between increasing the width and the height. We chose a heuristic approach based on the test cases we had seen, increasing the width by 3 and the height by 1 at each step.

The solution space created by the variables in this encoding is simple and obvious, but also much too large given the structure and constraints of the problem. Indeed, with this encoding, drawings with only about 30 nodes take hours to optimize.

This method naturally optimizes width/height first, since it needs to search for a width and height to make the SAT problem satisfiable. Bends can then afterwards be approximately optimized through the number of diagonal lines, by encoding a unary-encoded count of the number of diagonal lines and doing a binary search optimization on that number.

We describe this method here to illustrate a first, naive approach for solving the schematic drawing problem, and to highlight the importance of finding an efficient problem encoding. We have implemented this method and tested its performance compared to the other methods, as shown in Table 2.

3.4. Cross-section SAT encoding

Instead of directly representing a grid, we define a vertical cross-section c_k of the drawing, represented by a unary-encoded integer $y^k_{e_i}$ capturing the height of each edge e_i at some horizontal location in the drawing. This naturally allows us to use the edge vertical order $<_E$ as constraints on unary numbers $y^k_{e_i} <_E y^k_{e_j}$. Each pair of successive nodes is transformed into a sequence of such cross-sections, and we associate a direction $d^k_{e_i} \in \{\text{Up}, \text{Straight}, \text{Down}\}$ with each edge e_i at each cross-section c_k, giving the shape of the edge to the *left* (lower x value) of the cross-section. Cross-sections can be enabled or disabled (represented by b_k) to optimize the width of the drawing. Finally, the *ahead* Boolean $a^k_{e_i}$ for each edge at each cross-section marks whether the shape of the edge has already been constrained for the next cross-section to the right (higher x value), which allows nodes to impose edge shape constraints in both x-axis directions.

With this representation, we can impose constraints as follows:

1. Edge vertical order:

   $$(e_i <_E e_j) \Rightarrow \bigwedge_{c_k} y^k_{e_i} \leq y^k_{e_j}$$

2. A begin node at cross-section c_k constrains the edge shape to the right, and makes the y value unequal to the y value of other edges $e_j \in c_k$.

   $$a^k_{e_i} \wedge d^k_{e_i} = \text{Straight}, \quad \bigwedge_{e_j \in c_k} y^k_{e_i} \neq y^k_{e_j},$$

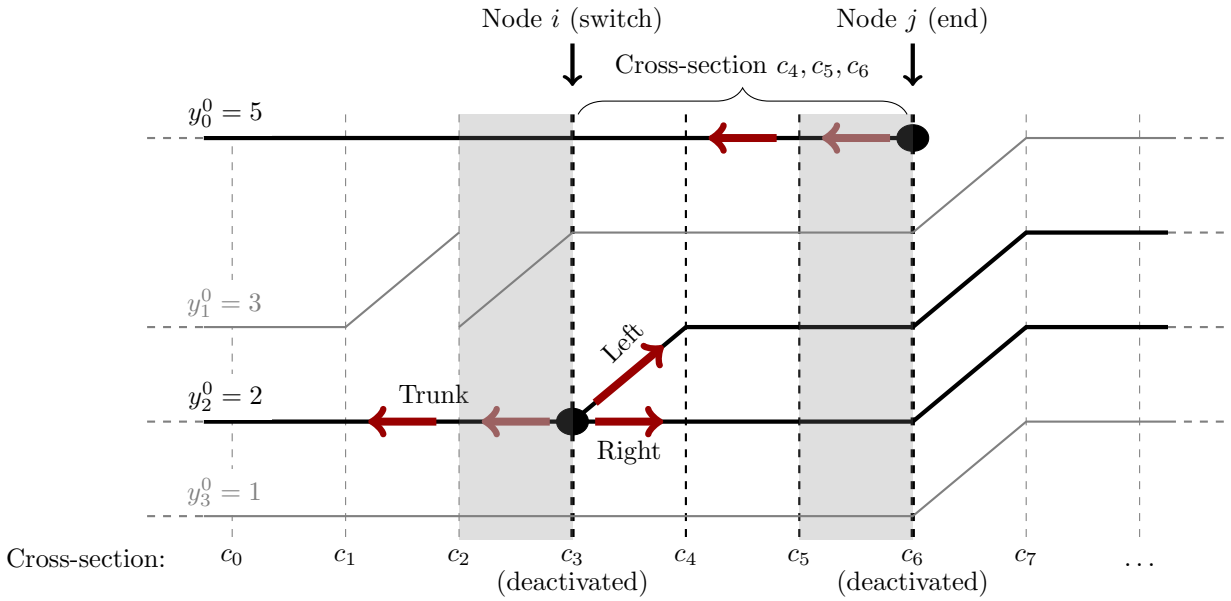


Fig. 10. Cross-section SAT representation. Dashed vertical lines show cross-sections c_i. Edges have a y value and a direction to the left of each cross-sec. Thick red arrows are constraints imposed by node type. Gray columns correspond to deactivated cross-sections, where shape constraints are propagated to the next or previous column.

and similar for end nodes, in the opposite direction:

$$\neg a^k_{e_i} \wedge d^k_{e_i} = \text{Straight}, \qquad \bigwedge_{e_j \in c_k} y^k_{e_i} \neq y^k_{e_j}.$$

3. A switch node at cross-section c_k constrains the edge shape in both directions by constraining the incoming edges e_i according to the node class variant. For example, for an outgoing left switch we have one incoming edge e_{i1}:

   $$\neg a^k_{e_i1} \wedge d^k_{e_i1} \neq \text{Up}$$

   The incoming edges e_i are replaced by the outgoing edges e_j in the cross-section representation. For example, for an outgoing left switch (see Figure 10) we have two outgoing edges e_{j1}, e_{j2} as the left and right ports, respectively:

   $$a^k_{e_{j1}} \wedge a^k_{e_{j2}},$$

   and we have two choices of shape:

   $$\left(d^k_{e_i} = \text{Straight}\right) \Rightarrow \left(y^k_{e_i} = y^k_{e_{j2}} \wedge d^k_{e_{j2}} = \text{Straight} \wedge d^k_{e_{j1}} = \text{Left}\right)$$

   $$\left(d^k_{e_i} = \text{Down}\right) \Rightarrow \left(y^k_{e_i} = y^k_{e_{j1}} \wedge d^k_{e_{j2}} = \text{Down} \wedge d^k_{e_{j1}} = \text{Straight}\right)$$

   Constraints are similar for other node classes.
4. Disabled cross-sections propagate all their values:

   $$\neg b_k \Rightarrow \bigwedge_{e_i \in c_k} \left\{y^k_{e_i} = y^{k+1}_{e_i} \wedge a^k_{e_i} = a^{k+1}_{e_i} \wedge d^k_{e_i} = d^{k+1}_{e_i}\right\}$$

5. Enabled cross-sections require consistency between edge shapes and y values:

   $$b_k \Rightarrow \bigwedge_{e_i \in c_k} \left\{\left(\neg a^k_{e_i} \wedge d^{k+1}_{e_i} = \text{Up}\right) \Rightarrow y^k_{e_i} + 1 = y^{k+1}_{e_i}\right\}$$

And correspondingly for *Straight* and *Down* directions.

6. Enabled cross-sections realize rightward-constrained *ahead* values a:

$$b_k \Rightarrow \bigwedge_{e_i \in c_k} \left\{ \left(a^k_{e_i} \Rightarrow y^k_{e_i} = y^{k+1}_{e_i}\right) \wedge \left(a^k_{e_i} \Rightarrow d^k_{e_i} = d^{k+1}_{e_i}\right) \wedge \neg a^{k+1}_{e_i} \right\}$$

With this formulation we can choose freely between prioritizing width, height, or bends, and the resulting plans have lower total width than for the level-based methods, since the grid-based method has the added freedom of inserting bends at any location along an edge. See Figure 14 for a comparison.

3.5. Level-based SAT encoding

We reformulate the problem using variables from the Boolean and bounded integer domains. Since we are dealing with small integers, we can transform the problem into a Boolean satisfiability problem (SAT) by encoding numerical variables into Boolean variables and use incremental SAT solvers which can be efficient for lexicographical optimization on small discrete domains, as ours.

Integers can be encoded into SAT in various ways. Eager encodings represent numbers and constraints directly using a set of Boolean variables and constraints and creates an equisatisfiable SAT instance. Most commonly used is the binary encoding (one Boolean for each bit) and the unary encoding (one Boolean for each distinct number). See [5] for details. Lazy encodings, as used in SMT solvers (see [3, 15] for an introduction), can avoid some of the work of transforming and solving a large SAT problem by abstracting the numerical constraints into marker Boolean variables. Only when the SAT solver sets markers to true, another procedure (the *theory solver*) will go to work on the numerical constraints and report unsatisfiable combinations back to the SAT solver.

Although the SAT problem itself does not directly concern numbers, much less numerical optimization, an *incremental* interface to a SAT solver allows solving many similar problems consecutively. For a set of constraints ϕ, we can perform numerical optimization on some number x by solving the sequence of formulas $\phi \wedge (x < m_1)$, $\phi \wedge (x < m_2)$, ..., where the sequence m_i is a linear or binary search over the range of x, locating the smallest value that satisfies the constraints. Querying the solver successively with such similar formulas incrementally is much faster than solving the instances separately.

We used the MiniSat [11] solver v2.2.0 with unary encoding of bounded integers and also lazy representation of unbounded integers with difference constraints, i.e. constraints of the form $x_i - x_j \leq k$, where k is a constant. Difference constraints are suitable as a first-line refinement in SMT solvers (see e.g. [6]) because they can be efficiently solved.

We keep the assumption from the previous subsection that each edge is assigned to a single level, and extend the problem representation as follows:

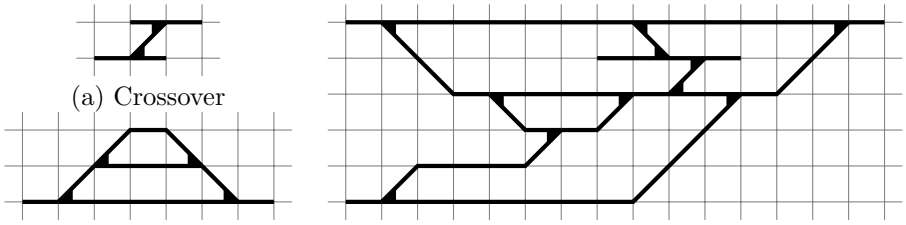
1. Distances between nodes are represented as a saturating unary number of size 2, i.e. $\Delta x \in \{0, 1, \geq 2\}$. This allows us to distinguish between short ($\Delta x \leq 1$) and long ($\Delta x \geq 2$) edges.
2. For each edge e_j, we use Booleans q^{up}_j and q^{down}_j to indicate a short edge pointing up/down, respectively, seen in the direction of increasing x.
3. Node vertical coordinates y_i and edge levels l_j are represented by unbounded integers on which we can conditionally impose difference constraints.
4. Variant selection $r_i \in R(c_j)$ for each node i indicates the node's variant from the available shapes $R(c_j)$ of the node class $c_j \in C$ listed in Figure 4.
5. Edge direction values, $d^{\text{begin}}_i, d^{\text{end}}_i \in \{\text{Up}, \text{Straight}, \text{Down}\}$, for the beginning and end of each edge e_i, are based on node variant values.

We need the following constraints:

- Each edge must be at least 1 unit long on the x axis.
- Edge ordering constraints for $e_a <_E e_b$:

$$l_a \leq l_b, \quad \left(\neg q^{\text{up}}_a \wedge \neg q^{\text{down}}_b\right) \Rightarrow l_a + 1 \leq l_b$$

If an edge is a short edge (such as a crossover between two adjacent tracks) it does not require its own



(a) Crossover

(b) Ladder sidings (c) Real-world example: Eidsvoll railML

Fig. 11. Output examples for the level-based SAT method. The crossover (a) requires only a 1 unit diagonal edge (improving Figure 8(b)). The ladder sidings (b) now use diagonal switch variants to improve width, height, and bends (improving Figure 8(d)). The Eidsvoll station (c) demonstrates real-world infrastructure imported from railML.

level, and we use instead the same level as the one of its end nodes which has the *highest* value. This allows to produce a better crossover drawing, as in Figure 11(a) instead of Figure 8(b).

- An edge i is short (q^{up} or q^{down}) if both ends have the same direction and the vertical distance between nodes is one:

  $$q_i^{\text{up}} \Rightarrow (d_i^{\text{begin}} = \text{Up}) \wedge (d_i^{\text{end}} = \text{Up}) \wedge (y_a + 1 = y_b)$$

  $$q_i^{\text{down}} \Rightarrow (d_i^{\text{begin}} = \text{Down}) \wedge (d_i^{\text{end}} = \text{Down}) \wedge (y_a - 1 = y_b)$$
- Direction on edge i decides vertical level constraints:

  $$(d_i^{\text{begin}} = \text{Straight}) \Rightarrow (y_a = l_i), \quad (d_i^{\text{begin}} = \text{Up}) \Rightarrow y_a + 1 \leq l_i,$$

  $$(d_i^{\text{begin}} = \text{Down}) \Rightarrow ((q_i^{\text{up}} \Rightarrow (y_a \geq l_i)) \wedge (\neg q_i^{\text{up}} \Rightarrow (y_a \geq l_i + 1)))$$

  And correspondingly for d^{end}.
- The sum of Δx values over the edge must match the shape of the edge:

  $$\left(q^{\text{up}} \vee q^{\text{down}}\right) \Rightarrow \Sigma_{j \in (a,b)} \Delta x_j \leq 1$$

  $$\left(\neg q^{\text{up}} \wedge \neg q^{\text{down}} \wedge \left(d^{\text{begin}} \neq \text{Straight} \vee d^{\text{end}} \neq \text{Straight}\right)\right) \Rightarrow \Sigma_{j \in (a,b)} \Delta x_j \geq 2$$

Since the shape of an edge is now explicit through d^{begin} and d^{end}, we can optimize for the number of bends to produce Figure 11(b) instead of Figure 8(d).

3.6. Symbols and labels

Railway engineering schematics also include a considerable number of symbols and labels, e.g., in a railway signaling schematic, signals and related equipment are drawn on the schematic, and the diagram is richly annotated with traveling distances, area limits, comments, and more. It is quite common that the track layout must be drawn in a way that accounts for the amounts and sizes of symbols and labels (see Figure 12). In consequence, we include in our tool (presented in Section 4) options for placing symbols on two rows above and below each track, which is suitable for a commonly used schematic signaling drawing where trackside signaling equipment is drawn on and around the tracks. We present here the method of doing this using difference constraints, which can thus be easily combined with the method of drawing the tracks.

The level-based SAT encoding already relies on difference constraints for solving the track layout, so for ensuring that the drawing has enough space for all the symbols, we can straight-forwardly encode the symbol placement constraints from Section 2.4 using difference constraints. We also limit the number of bands to *two* bands above and below the tracks, and shorten the bands at a fixed offset from switches to avoid symbols overlapping, see Figure 13. Since we are using integer domains for the difference constraints, the scale that we have used above to describe the model can be too coarse for placing many small symbols. We scale up the X axis of the difference constraints by a factor of 10000, and then straightforwardly translate the constraints:

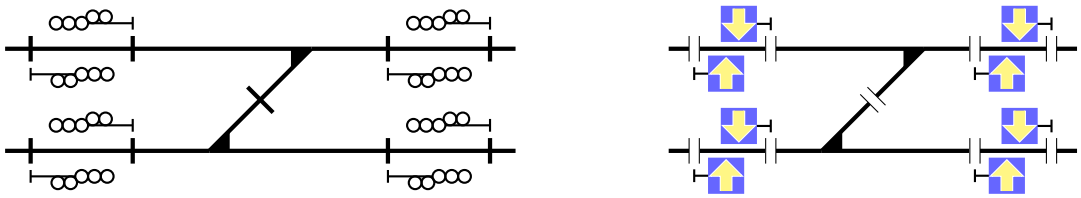


Fig. 12. Symbol placement and style may affect track layout.

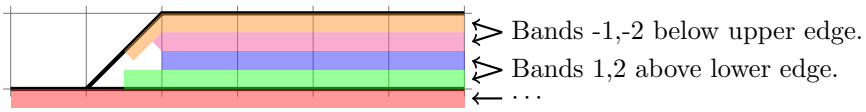


Fig. 13. Label placement can be done by restricting symbols to fit into a set of bands above and below each line, which reduces the constraints to linear ordering.

- Ordering symbols by linear reference position, for consecutive symbols a and b:

  $$x_c^a \leq x_c^b,$$

  where x_c^i is the center point for symbol $i \in \{a, b\}$.
- Ordering symbols within the same band, for consecutive symbols a and b:

  $$x_r^a \leq x_l^b.$$

- For a pair of symbols a and b with an alignment constraint:

  $$x_c^a \leq x_c^b \wedge x_c^b \leq x_c^a.$$

However, the optimization task of minimizing the symbols' offsets from their proportional position is not readily expressible in SAT modulo difference constraints. We suggest two approaches to solve this problem:

1. Use a full-featured SMT solver with optimization capabilities, such as Z3 or OptiMathSAT.
2. After ensuring satisfiability with the SAT modulo difference logic solver, transfer the symbol constraints to a linear programming solver, such as COIN-OR CLP/CBC, to optimize their final positions while keeping the track plan fixed.

To avoid installation, distribution and licensing issues, we avoided both approaches in our tool, and went instead for a heuristic approach: setting the starting positions for all symbols to their proportional value before solving the difference logic constraints ensures that symbols are not moved from the proportional location unless forced to by a constraint. This approach is not guaranteed to produce optimal solutions, but has worked well on our examples.

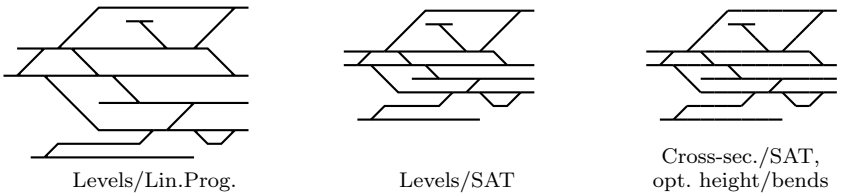
3.7. Quality and performance evaluation

We have implemented and compared the performance of the four methods described above, summarized in Table 2. The Direct/SAT encoding's performance is too poor to be of practical value. The Levels/SAT encoding is the fastest, and produces good output when optimizing for bends first. Cross-sec./SAT is slower, but is more capable for optimizing for height and width.

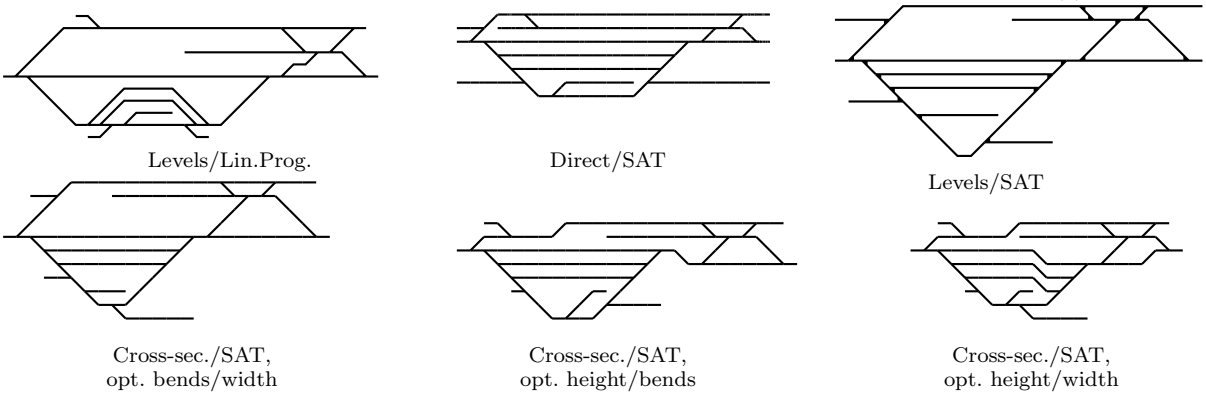
In Table 3, we summarize the quality aspects discussed in the subsections above for each of the methods. We classify the methods' ability to attain the given criteria into the following categories:

- "poor" if the method produces obviously sub-optimal results,
- "adequate" if the method optimizes approximately and may produce sub-optimal results, but still works well in practice,
- "optimal" if the method produces optimal results.

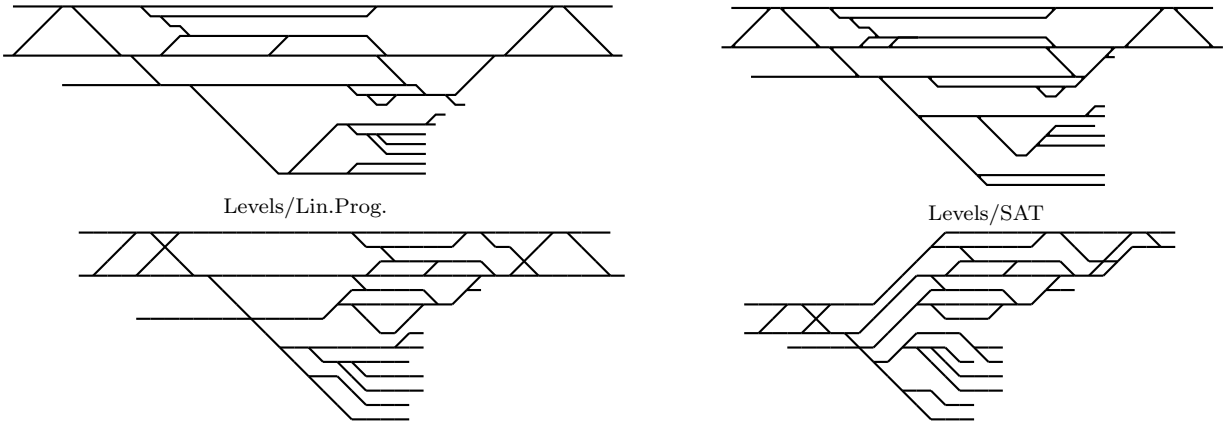
(a) Drawings of *Eidsvoll station*, infrastructure imported from BaneNOR railML [21]. In this infrastructure there is little reason to rotate switches, so the linear programming model produces good results except for the excessive height. The levels SAT improves the height by using short crossover tracks. The cross-section SAT model attains a smaller drawing by deviating from the "levels" assumption described in Section 3.2. The direct SAT drawing does not respect linear order, and it makes a very compressed drawing.



(b) Drawings of *Asker station*, imported from [21]. Like in (a), there is little reason to rotate switches, all approaches produce similar drawings, though the linear programming model creates excessive height for the same reason as in (a).



(c) Drawings of *Arna station* ongoing construction project, infrastructure imported from a RailCOMPLETE CAD project. The linear programming model produces the unwanted ladder style described also in Figure 8. Note that the direct SAT model drawing does not respect linear order. The cross-section SAT model optimizing for bends first is very similar to the levels SAT model (which also optimizes for bends first), but the cross-section SAT model can also be used to attain even lower height and width by optimizing for these first and deviating from the "levels" assumption described in Section 3.2.



(d) Drawings of *Weert station*, infrastructure remodeled from figures in [7]. We see that the linear programming model has much wasted space. The levels SAT is somewhat better, but on this infrastructure, the cross-section SAT method is clearly better because it is able to pack together the fan-out shapes and not reserve excess space for edges. Note that some method implementations did not include crossings, so these have been left out of some drawings.

Fig. 14. Comparison of three optimization models on various infrastructure models: Levels/Lin.Prog. (see Sec. 3.2), Direct/SAT (see Sec. 3.3), Cross-sec./SAT (see Sec. 3.4), Levels/SAT (see Sec. 3.5).

Model	Source	Size	Direct/SAT		Levels/SAT		Cross-sec./SAT			
			hwb	size (v/c)	bhw	size (v/c)	hwb	hbw	bhw	size (v/c)
Eidsvoll	[21]	35	60.7	57k/153k	0.02	2.3k/0.7k	0.05	0.06	0.33	4.0k/28k
Arna	RC	57	294	167k/493k	0.03	4.9k/1.3k	0.26	0.65	1.06	11k/100k
Asker	[21]	64	T/O	104k/295k	0.04	5.6k/2.0k	0.61	1.02	0.87	14k/124k
Weert	[7]	102	T/O	304k/969k	0.18	11k/4.0k	0.72	19.3	21.4	29k/327k
5x10	T	228	T/O	2.8M/13M	0.58	35k/2.7k	5.83	7.48	8.08	46k/364k
5x20	T	478	T/O	2.8M/12M	3.37	97k/7.7k	279	299	T/O	265k/4.2M
10x5	T	203	T/O	3.0M/14M	0.40	28k/2.0k	0.52	0.59	1.08	20k/83k
20x5	T	403	T/O	3.0M/14M	1.73	70k/4.0k	1.95	2.50	3.36	44k/165k
10x10	T	453	T/O	2.6M/12M	2.74	86k/5.5k	21.9	22.4	40.7	96k/727k
15x15	T	1053	T/O	2.3M/10M	22.7	255k/15k	T/O	T/O	T/O	N/A

Table 2. Running times in seconds on a mid-range workstation. Time-outs (T/O) indicate exceeding 300 s. Model sizes are given as the sum of the number of nodes and edges. Models were obtained from BaneNOR [21], a RailCOMPLETE CAD project (RC), and adapted from [7]. Scaling test models (T) named $n \times m$ consist of n serially connected stations, each spreading out to m parallel tracks. Optimization criteria are height (h), width (w) and bends (b). The size columns show the number of SAT variables and clauses (v/c).

		Optimization criteria			
Model	**Hard constraints**	Width	Height	Diagonals	Bends
Levels/Linear programming	✓	poor	adequate	poor	poor
Direct/SAT	✓	optimal	optimal	optimal	optimal
Levels/SAT	✓	adequate	optimal	adequate	adequate
Cross-section/SAT	✓	optimal	optimal	optimal	optimal

Table 3. Quality comparison for the methods. The Direct/SAT and Cross-section/SAT methods optimize all criteria exactly, but can take too long to compute for practical usage. The Levels/Linear programming method is highly scalable, but suffers from poor quality in width, diagonals, and bends. The Levels/SAT method offers a compromise where width, diagonals and bends are optimized, but according to the simplified "levels" edge definition which does not represent all possible drawings.

The qualitative differences between the methods are exemplified by the comparison in Figure 14. The linear programming model's excess length and height caused by crossovers (see Figure 8(b)) can be seen in each of the examples in Figure 14(a-d). The linear programming model's excess length caused by ladder sidings (see Figure 8(d)) can be seen in the Arna and Weert examples in Figure 14(c-d). The effect of the "levels" assumption described in Section 3.2 can be observed by comparing the levels SAT method to the cross-section SAT method in Figure 14(a,c,d). Note that the direct SAT method drawings shown in Figure 14(a,c) do not respect the linear ordering constraints, as they were produced using an earlier version of the input data format specification where the linear reference positions were not available.

We conclude that Cross-section/SAT and Levels/SAT are the only ones which have good enough running times and visual quality to be used in practice. The Cross-section/SAT with unbounded number of bends per edge is able to optimize drawing size further than the levels-based model. However, this comes at the cost of increased running time.

4. Software tool support for railway infrastructure design

Computer software tools can have an important supporting role in the complex design processes for railway infrastructure, both for engineering analysis and verification work, and for cross-discipline coordination and project management. The use of software for railway infrastructure engineering tasks has in many companies not advanced beyond computerized drafting. This section describes software that is in use in railway engineering, and how they could benefit from automating schematic infrastructure drawings.

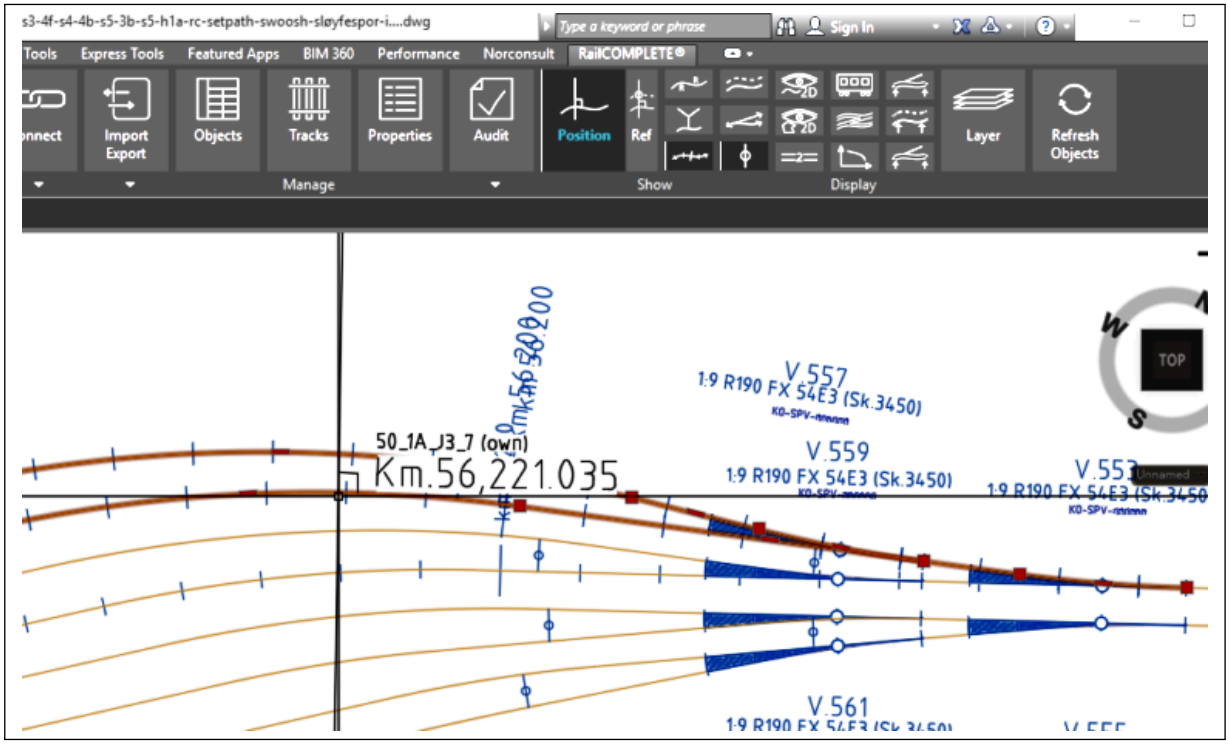


Fig. 15. The RailCOMPLETE CAD tools extends Autodesk AutoCAD with railway-specific drafting and modeling capabilities.

The main types of tools in use in railway engineering today are:

Drafting tools, often called CAD (computer-aided design/drafting), provide efficient means for producing geographical or schematic drawings of a construction project. Today, these are being extended to so-called BIM, Building Information Management, which typically means extending the CAD model to include 3D drawings of all disciplines for realistic visualization, and to include semantic data on components involved in the construction process.

Examples of general CAD programs in use for all sub-disciplines include Autodesk AutoCAD and Bentley MicroStation. For some disciplines, there are also domain-specific CAD programs available, most notably for railway track design which is supported by Trimble NovaPoint, AKG Vestra, Card 1, and more. Signaling and interlocking design is supported by WSP ProSig and RailCOMPLETE[3] (see Figure 15).

Databases and models of railway infrastructure and related information are used to store and transfer infrastructure data between companies, and between engineering, maintenance and operations sub-organizations. Typically, each national railway has a database model and a central database to store information about their railway network for engineering, maintenance, and operational planning. Examples include Ariane/Gaïa in the French SNCF railways, PlanProML in the German DB railways, and Banedata in the Norwegian Bane NOR railways.

Some efforts on international standards for data models are gaining traction, such as railML, RailTopoModel, EuLynx, and IFC, all of which are aimed at improving integration between software tools, and data exchange between infrastructure managers and contractors across different countries.

The railML format (see Figure 17 for an example) is an XML based language for data exchange of railway designs, developed by an international standardization committee, and was used as input to our tool as it is designed for exchange of infrastructure data between programs. railML consists of sub-schemas for time

[3] See the RailCOMPLETE web page: `http://railcomplete.no/`

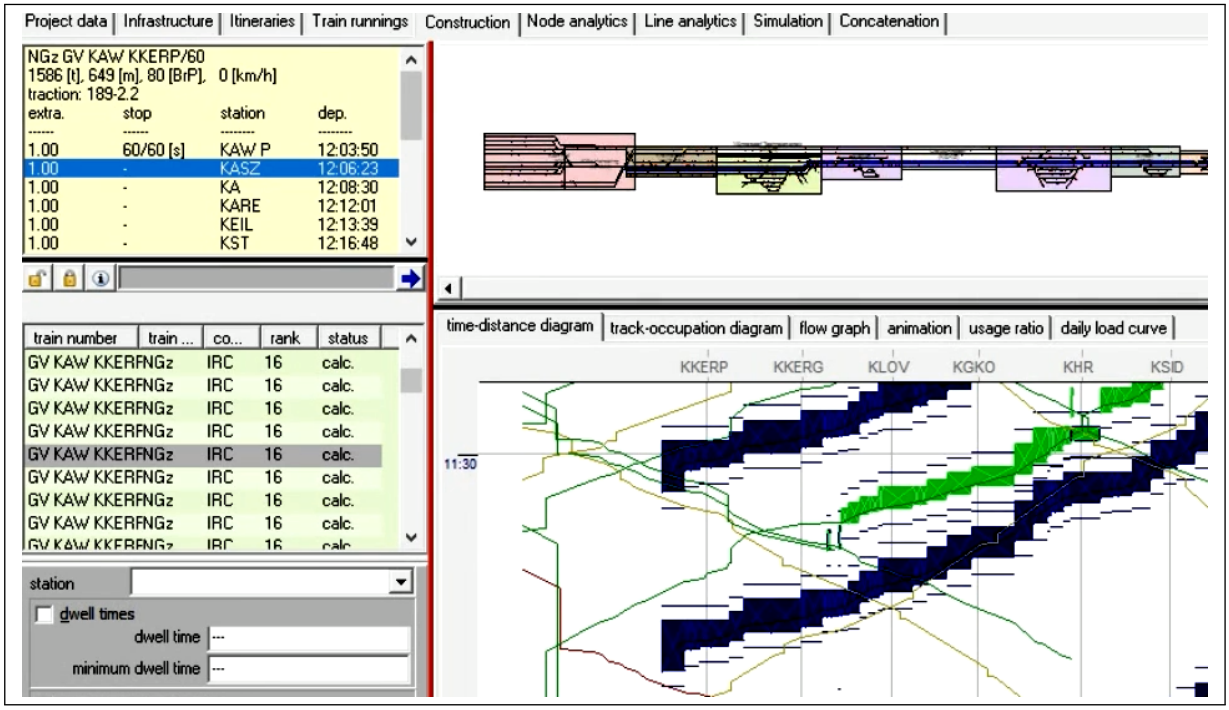


Fig. 16. The LUKS capacity analysis tool offers analytical models and simulation models on comprehensive infrastructure and time table data.

```
- <infrastructure id="inf01">
  - <tracks>
    - <track id="tr01" name="track a02" type="secondaryTrack" mainDir="none">
      - <trackTopology>
        - <trackBegin pos="0" id="tr01_tb">
            <bufferStop id="tr01_bs01"/>
          </trackBegin>
      - <trackElements>
        - <platformEdges>
            <platformEdge id="tr01_pe01" name="Gleis 2" xml:lang="de" pos="200" dir="up"
            height="550" length="200"/>
          </platformEdges>
        </trackElements>
      - <ocsElements>
        - <signals>
          - <signal id="tr01_si01" code="68N1" pos="450" absPos="450" dir="up" function=
            ocpStationRef="ocp02">
              <etcs switchable="false" level_2="true"/>
            </signal>
          </signals>
```

Fig. 17. The railML XML format is hierarchically structured, the infrastructure element contains tracks, and tracks contain trackside elements such as signals, and track geometry features such as radius and gradient.

table, rolling stock, and infrastructure. The infrastructure schema is organized with a list of tracks at the top level of the hierarchy. Tracks contain sub-elements for (1) movable track elements, such as switches, crossings, and derailers, (2) trackside elements such as signals, detectors, and balises, (3) track geometry, such as radius and gradient, and (4) operational status, such as country borders, electrification, platform adjacency, and much more. Ends of tracks, along with switches and crossings, are considered *nodes* which can be connected to each other by mutually cross-referencing each other by name (using the XML attribute `id`). The recent railML version 3 now also contains a sub-schema for interlocking specifications.

Analysis tools are typically specific to a sub-discipline. A major category of analysis tools is the capacity and time tabling tools. These tools use detailed data from the infrastructure, rolling stock, and time table domains to analyze changes to time table and stochastic effects of delays and congestion. Examples of capacity and time table analysis tools include VIA LUKS[4] (see Figure 16), OpenTrack, and RMCon RailSys.

Examples of other analysis domains include Sicat Master for catenary power line analysis, and Prover Trident for formal verification of interlocking implementations.

Automatic schematic drawing of railway infrastructure eliminates a major obstacle in transferring information between software from these categories. For example, extracting data from a database for use in analysis tools often requires drawing a schematic manually, and the resulting drawing is seldom reusable outside that specific analysis software. Another example is rapid iterations in early-stage construction projects, where models are first constructed in geometric CAD software, and then transferred to databases or analysis software. With automatic schematic drawings, this transfer is immediately available, avoiding manual modeling for a validation process which might be quite time-consuming or possibly skipped altogether.

5. `railplot`: a scriptable plotting tool for railway infrastructure

We have developed a tool with a command-line interface, called `railplot`, that can generate railway infrastructure drawings of the kind described and shown in this paper, available online[5]. The program is written in the Rust programming language, using the MiniSat v2.2 SAT solver through its library API.

The railway software industry does not have widely implemented standards for data representation and transfer, so a tool that works on railway infrastructure data needs to be flexible in how it reads and writes data to be applicable in different settings. Also, many applications require strict adherence to specific drawing styles which vary widely between different countries and companies. To accommodate such a variety of requirements, we considered the following approaches:

- A **programming library interface** could offer full flexibility of input/output formats by requiring the user to call the library with operations that correspond to the internal model of the automated drawing solver. In effect, this would require users of the software to program the required data transfer and drawing output routines themselves. `railplot` can be used as a library in this way, but we do not expect that railway engineers will directly take such a library into use, though it could be useful for software engineers in a development project for railway software.
- Using **standard file formats** is another approach to integrating with other railway software systems. `railplot` can be used in this way by using input from railML 2.x or a custom file format, and output given in SVG or a custom JSON structure. However, we have not defined a specific input language for declaring which railML objects are drawn as symbols with their size and graphic display. The tool can import railML files as track network input and track-side object symbols, or use a custom format for directly specifying topology. The tool offers two choices of built-in symbol appearances: "*simple*" for generic lamp-like signals and detector, and "*ERTMS*" for ERTMS-style marker boards and detectors (see Figure 12).

Both of the above approaches can work well for integrating a tool in many different settings, especially for dedicated software engineers. However, many engineers that work on railway and other infrastructure projects can be competent programmers, but not on a dedicated professional level where they are able and willing to invest time and effort into installing a development environment, reading API documentation, parsing from one data structure to another, and setting up all railway symbols from scratch.

To make it easier for non-professional programmers to customize the plotting tool, we have organized `railplot` so that a scripting language drives the main part of the program. This approach has been successful

[4] See the LUKS web page: `https://www.via-con.de/en/development/luks/`

[5] See the `railplot` web page: `https://github.com/luteberget/railplot`. The project is also archived at Software Heritage (`https://archive.softwareheritage.org/swh:1:snp:4f0ec805bcce1785abe2a0a31f330cb7ab8b6034;origin=https://github.com/luteberget/railplot/`) and Github Archive Program (`https://archiveprogram.github.com/arctic-vault/`).

in other plotting tools such as Gnuplot[6] and Graphviz[7]. We used the Lua programming language[8] and have embedded its interpreter into the `railplot` program. The user can export the application's default script to a Lua file and make smaller or larger customizations. The following are the main features of the scripting language interface:

- **Reading inputs**: the `railplot` program can load a railML (or other XML formats) and convert the XML structure into Lua objects. The plot script defines how the XML structure is interpreted into symbol objects. The objects should have the following properties: (a) `pos`, the linear reference position, (b) `width`, the width of the symbol in drawing units, (c) `origin`, the offset of the symbol center-point from the left as a fraction of the symbol width, and (d) `level`, which selects the band on the track (see Figure 5). An example using railML might look like this:

```
model = load_railml {
        filename = input_file,
        get_pos = function(o) return o.absPos or o.pos end,
        symbol_info = function(o)
            if o._elem == "signal" then
                level = o.dir == "up" and -1 or 1
                origin = o.dir == "up" and 0.0 or 0.4
                return { pos = o.absPos or o.pos,
                width=0.4, origin = origin, level = level }
    (...)
```

- **Drawing primitives**: the default script defines a function that maps from a symbol object into one or more drawing primitives. The primitives are lines, rectangles, and circles. The following example draws a train detector as a line crossing the track.

```
function draw_symbol(o)
    if o._elem == "trainDetector" then
        return line(0,-0.05,0,0.05)
    elseif o._elem == "signal" then
    (...)
```

  Each primitive is drawn in a coordinate system rotated and translated so that the point on the track it belongs to is at the origin and the positive direction of the track is along the positive direction on the x-axis. The user is also free to define new such primitives, for example image file references.
- **Output format**: finally, the script contains a function that maps from the drawing primitives to an output format. The following example defines outputting a rectangle to TiKZ format:

```
if output_format == "tikz" then
    function rect(x0,y0,x1,y1)
        return "\\draw (".."x0.."," .."y0.."") rectangle ("..x1.."," .."y1.."");"
    end
    (...)
```

  The tool's default script can produce output in JSON format (for custom visualization or post-processing), SVG (for use in web pages and web applications), or TikZ (for use in LaTeX documents).

Driving the program using an embedded scripting language solves the problem of needing to install a programming environment on the user's computer, but still allows full customization using a full-featured programming language.

The tool implements the Levels/SAT method described in Section 3.5, as we found this method to give the best trade-off between drawing quality and running time (see Table 2).

Implementations of the three other methods described in this paper are also available for download at the same address, but they are not integrated into the scriptable tool.

[6] See the `gnuplot` web page: `http://www.gnuplot.info/`
[7] See the `Graphviz` web page: `https://www.graphviz.org/`
[8] See the Lua web page: `http://www.lua.org/`

6. Conclusions and Future Work

We have demonstrated the feasibility of using an incremental SAT solver to automatically produce and optimize schematic railway drawings using several different optimization criteria. However, the choice of encoding makes a significant difference in the size of models that can be handled in a reasonable amount of time, cf. Table 2. The direct representation using an explicit grid fails to handle instances of relevant scale. Only after reformulating the problem in a more structured solution space, where the order of symbols is hard-coded into the problem, rather than added as a constraint after the fact, we were able to solve industrial-size instances in reasonable time for interactive use (i.e., under 1s). A remaining interesting problem is the study of the inherent computational complexity of the linear schematic drawing problem.

Our goal is that professionals should be able to rely on high-quality automatic schematics, which requires further tailoring of symbol and text placement to specific use cases, and integration with GUI tools.

Acknowledgments:

We would like to thank Koen Claessen and Martin Steffen, as well as engineers from Railcomplete AS, for their help with this work, which was partly funded by Railcomplete AS and the Norwegian Research Council through the project *RailCons – Automated Methods and Tools for Ensuring Consistency of Railway Designs*.

References

[1] IRS 30100: RailTopoModel - railway infrastructure topological model. The International Union of Railways (UIC), 2016.

[2] S. Avelar. *Schematic Maps on Demand - Design, Modeling and Visualization*. PhD thesis, ETH Zürich, 2002.

[3] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] M. Björk. Successful SAT encoding techniques. *JSAT*, 7(4):189–201, 2011.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 317–333. Springer, 2005.

[7] A. Brands. Automatic generation of schematic diagrams of the Dutch railway network. M.Sc. thesis, Radboud University, 2016.

[8] S. Cabello, M. de Berg, and M. J. van Kreveld. Schematization of networks. *Comput. Geom.*, 30(3):223–228, 2005.

[9] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235–282, 1994.

[10] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT conference 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[12] D. Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen*. PhD thesis, ETH Zurich, 2002.

[13] B. Luteberget, K. Claessen, and C. Johansen. Design-time railway capacity verification using SAT modulo discrete event simulation. In *FMCAD*. IEEE, 2018.

[14] B. Luteberget, K. Claessen, and C. Johansen. Automated Drawing of Railway Schematics Using Numerical Optimization in SAT. In W. Ahrendt and S. L. T. Tarifa, editors, *15th International Conference on Integrated Formal Methods (iFM 2019)*, volume 11918 of *LNCS*, pages 341–359. Springer, November 2019.

[15] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[16] M. Nöllenburg and A. Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Tr.Vis.Comput.Graph.*, 17(5):626–641, 2011.

[17] M. Nöllenburg. Automated drawing of metro maps. Technical Report 25, Universität Karlsruhe, Karlsruhe, 2005.

[18] O. Oke and S. Siddiqui. Efficient automated schematic map drawing using multiobjective mixed integer programming. *Computers & OR*, 61:1–17, 2015.

[19] M. M. Ozdal. *Routing Algorithms for High-Performance VLSI Packaging*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[20] A. Papakostas and I. G. Tollis. Algorithms for area-efficient orthogonal drawings. *Comput. Geom.*, 9(1-2):83–110, 1998.

[21] Bane NOR: Model of the Norwegian rail network. `http://www.banenor.no/en/startpage1/Market1/Model-of-the-national-rail-network/`, 2016.

[22] S. Seyedi-Shandiz. Schematic representation of the geographical railway network used by the Swedish transport administration. M.Sc. thesis, Lund University, 2014.
[23] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
[24] T. C. van Dijk, F. Lipp, P. Markfelder, and A. Wolff. Computing Storyline Visualizations with Few Block Crossings. In *GD*, pages 365–378. Springer, 2018.
[25] A. Wolff. Drawing subway maps: A survey. *Inf. Forsc. Entwick.*, 22(1):23–44, 2007.