

Show Me How It's Wrong

Counterexample Visualisation in Static Railway Verification

Victor Cai

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Master of Science by Research



Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2024-11-18



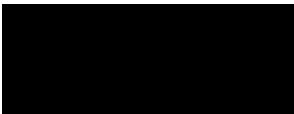
Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed  (candidate)
Date 2024-11-20

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed  (candidate)
Date 2024-11-20

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed  (candidate)
Date 2024-11-20

This thesis is dedicated to the countless dedicated individuals in the railway industry, whose hard work ensures that our trains continue to be the safest, most efficient, and most sustainable mode of transport on land.

This thesis and its associated software were produced entirely by a human and do not contain any AI-generated content.



Abstract

The introduction of modern signalling and control systems on the railway requires that the infrastructure topology data be verified against a number of design constraints to ensure safety and correctness. At ██████████ ██████████ a software toolchain has been developed to automate this verification process, based on a formal proof tool and its language. In this thesis, we present an enhancement to this toolchain introducing the functionality to graphically display counterexamples. The requirements for this extension is established through user engagement in the form of a focus group study, and its implementation is carried out across several industrial-sized software projects. The work involves human-computer interaction design, formal methods, and the development of data formats as well as programming.

Acknowledgements

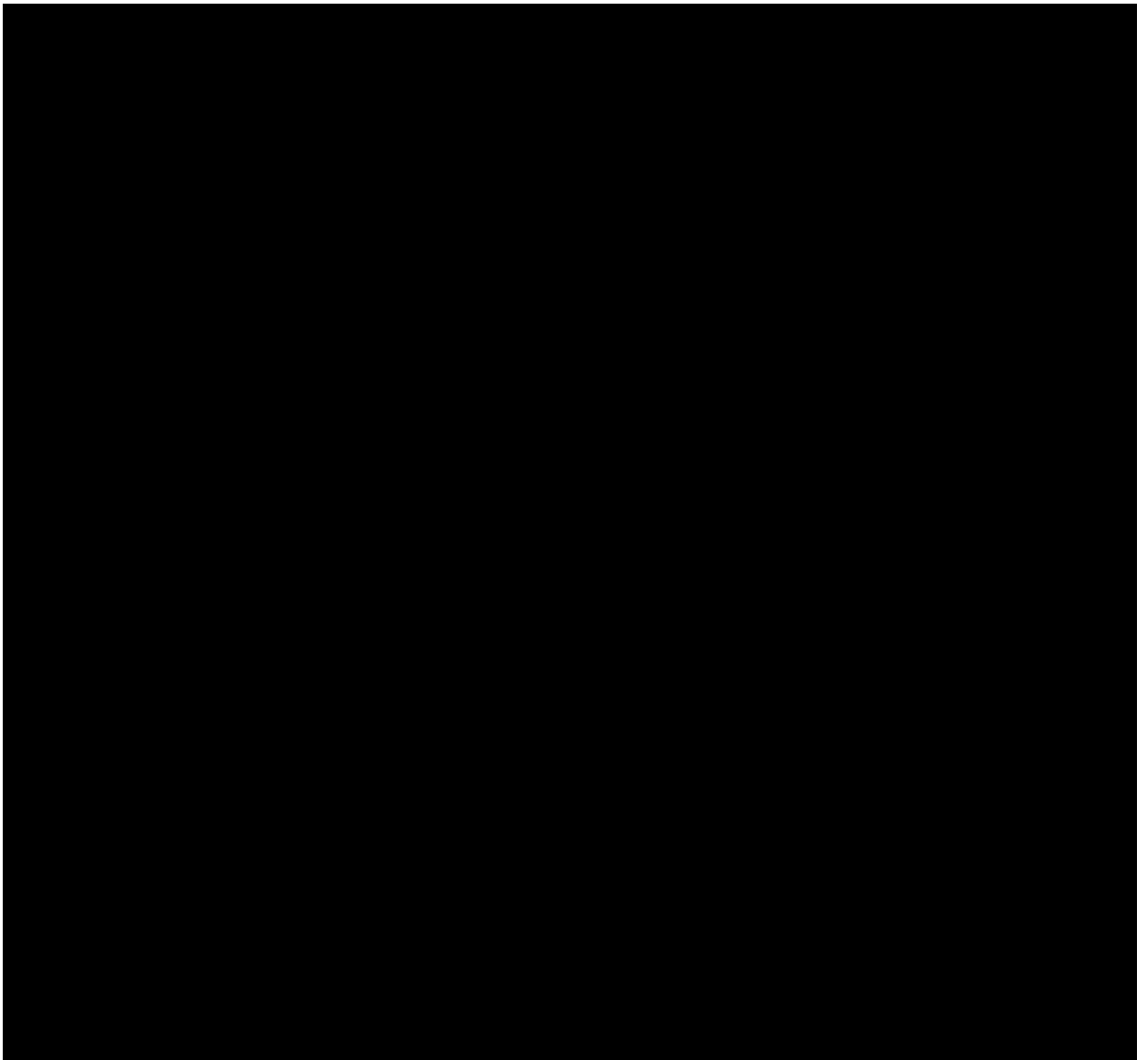


Table of Contents

List of Figures	vii
1 Introduction	1
1.1 Our Contributions	3
1.2 Presentations and Publications	3
1.3 Overview of the Thesis	4
I Background	5
2 Related Work	7
2.1 Domain-Specific Visualisation	7
2.2 Railway Infrastructure Verification	8
3 Railway Concepts	9
3.1 Infrastructure Elements	9
3.2 Design Rules	13
4 Underlying Proof Technology	17
4.1 Satisfiability Modulo Theories	18
4.2 Z3 and SMTLib2: An Introduction by Example	19
4.3 Workflow of the XXXXXXXXXX	22
5 Software Tools and Technologies	25
5.1 Data Formats	25
5.2 Programs and Libraries	30
II Contribution	33
6 Designing the Visualisations: Developing Requirements	35
6.1 Establishing Requirements: Our Focus Group Study	35
6.2 Our Proposed Designs for Representing Single Errors	47
7 Software Architecture and Design	53

7.1	Overall Process	56
7.2	Design Considerations	60
8	Modifications to the [REDACTED]	65
8.1	Adding the Counterexample Pass to the Verification Algorithm	67
8.2	Extracting Information from the Unsat Core	70
8.3	Constructing Counterexample Data	73
8.4	Saving Counterexample Data	79
9	Implementing the Visualisations in [REDACTED]	85
9.1	Modifying the [redacted] Schema	93
9.2	Preparatory Work on the Codebase	97
9.3	Defining the Appearance of Counterexample Elements	102
10	An Extended Example of Counterexample Visualisation	107
III	Conclusion	113
11	Summary	115
12	Future Work	117
12.1	Near Term Improvements	117
12.2	The Next Project	118
12.3	Possible Future Development in Counterexample Visualisation	119
	Bibliography	121
	Appendices	127
A	Glossary of Railway Terms	129
A.1	General Railway Terms	129
A.2	Terms Specific to [REDACTED]	131
B	Files Demonstrating Data Formats	133
C	Source Code of the Z3 Toy Example	171
D	Focus Group Material	175
D.1	Briefing Documents	175
D.2	Briefing Presentation	180
D.3	Exercise Sheets	190
D.4	Completed Exercise Sheets	196
D.5	Transcript of Plenary Discussion	214
E	Files Involved in the Extended Example	229

List of Figures

1.1	An abstract illustration of the generic process of verifying a system against a property, with the focus of our project highlighted.	1
1.2	An overview of the scheme plan data verification toolchain at [REDACTED] with our extensions highlighted.	2
3.1	An excerpt of a scheme plan for a railway line near Bristol, courtesy of Network Rail.	9
3.2	An extract of the Network Rail standard on scheme plan symbols [66].	10
3.3	Change of ELR shown at a junction on the scheme plan.	10
3.4	A simplified diagram of the triangle formed by the WEY, SWY, and WYL lines near Westbury station.	11
3.5	Signal BL1804, located at the end of platform 1 at Stapleton Road station.	11
3.6	The UFR track at Stapleton Road represented in the Node-Edge Model.	12
3.7	Diagram from [84] illustrating the requirements of design rule BG-03.	15
4.1	An overview of the [REDACTED]'s workflow.	17
4.2	The scheme plan used in the SMT solving example.	19
5.1	[REDACTED] data for balises and balise groups displayed in [REDACTED] showing how each balise contains a reference to its balise group.	27
5.2	The "[REDACTED]" tab of the [REDACTED] user interface.	30
6.1	The scheme plan used in the focus group examples.	37
6.2	Example of an exercise sheet used in the focus group study.	38
6.3	Design presented by group 1 for task 1.	40
6.4	Design presented by group 2 for task 1.	41
6.5	A step in the workflow sequence designed by group 3.	42
6.6	Group 1's response to task 3.	44
7.1	Sequence of messages between the tools when the [REDACTED] returns a <i>Pass</i> or <i>Unknown</i> result. We use the UML alt operator to denote possible outcomes of the same action. Note that the <code>getreport</code> command will only be issued (and retrieve one of the possible verification results) if the job has completed.	54
7.2	Sequence of messages between the tools when the [REDACTED] returns a <i>Fail</i> result. Our modifications are shown in red.	55

7.3	The overall transformation process for counterexample data.	58
7.4	The data types we add to the [redacted]'s report for counterexamples.	59
7.5	The data types we add to [redacted] ([redacted] files).	59
8.1	An overview of the server-side operations involved in producing counterexample data.	66
8.2	The [redacted]'s verification algorithm after our modifications.	67
9.1	The transformations between data models involved in the process of graphically displaying a set of [redacted] data in [redacted].	86
9.2	The [redacted] objects, each representing a row in a [redacted] file.	86
9.3	Attributes of a [redacted] object that represents a track.	87
9.4	Data parsed from [redacted] files shown as tables in [redacted].	88
9.5	The RMTrack representing the first track in Listing 5.2.	90
9.6	A segment of the example track represented in models used for rendering the track layout diagram.	92
10.1	The scheme plan used in our extended example, as displayed in [redacted].	107
10.2	Infrastructure data from [redacted] files displayed in [redacted] as tables.	108
10.3	The "[redacted]" tab of [redacted] prior to starting verification.	108
10.4	The "[redacted]" tab of [redacted] after starting verification.	109
10.5	Verification results displayed in the "[redacted]" tab of [redacted].	110
10.6	The counterexample displayed in graphical form.	111
10.7	Counterexample data from our additional [redacted] files, shown in tabular form.	111
A.1	The principal part of a set of points, from [29].	129

Chapter 1

Introduction

Contents

1.1	Our Contributions	3
1.2	Presentations and Publications	3
1.3	Overview of the Thesis	4

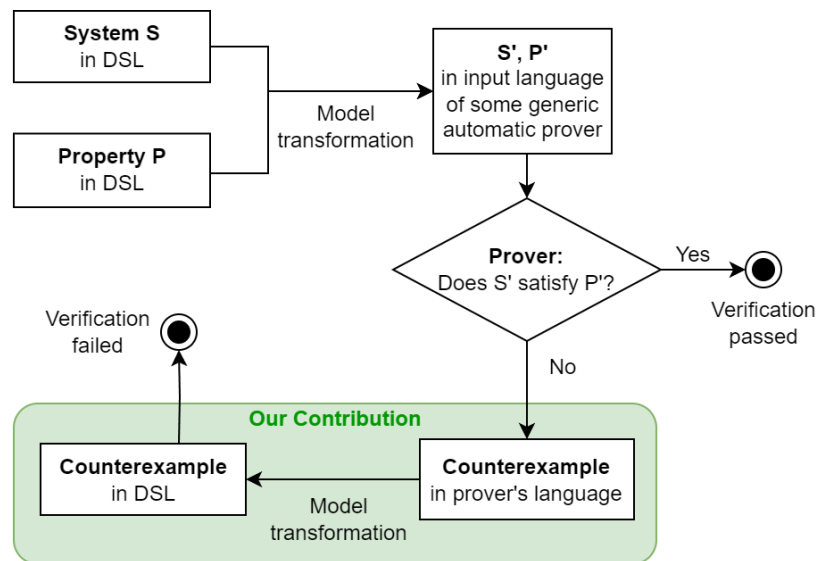


Figure 1.1: An abstract illustration of the generic process of verifying a system against a property, with the focus of our project highlighted.

Formal Methods (FMs) help address hardware and software engineering challenges by precisely expressing requirements and supporting analyses to increase confidence in the correctness of the system in question. In contrast with mathematical theories, however, for FMs to be applicable to industrial-scale systems — which often involve large numbers of axioms,

relatively frequent changes to these axioms, and intellectual property restrictions — tool support is needed [79].

FMs often use Domain-Specific Languages (DSLs) as an interface to the application domain — examples of work on DSLs include [94, 34, 43]. For verification, these DSLs are translated into an input language of a general-purpose proof tool. In case the proof tool finds a counterexample, this counterexample is presented in the language of the tool but not in the DSL. This output requires further interpretation to be useful for domain experts working to refine the system being verified.

Our project sets out to provide a translation from the tool language into a DSL, as illustrated in Figure 1.1.

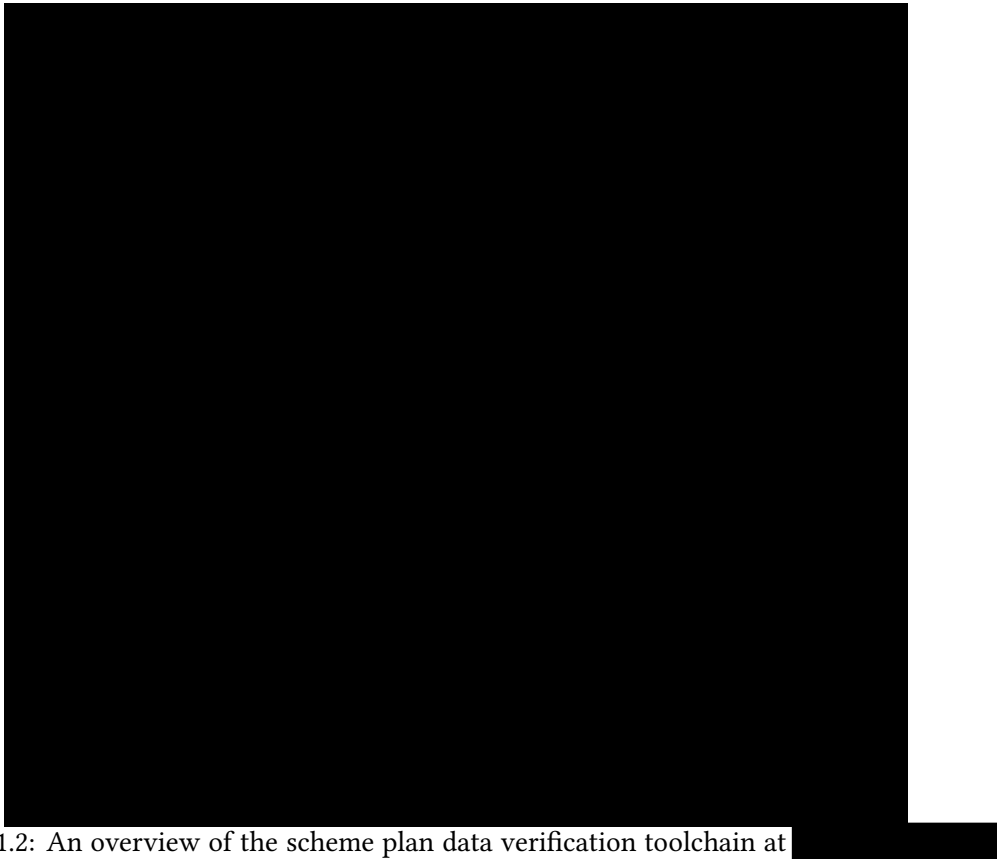


Figure 1.2: An overview of the scheme plan data verification toolchain at [redacted] with our extensions highlighted.

In the railway domain, *scheme plans* are used to encode infrastructure topology and placement of equipment. In modern railway signalling systems like the European Train Control System (ETCS) [19], the correct working of safety-critical control computers requires that scheme plans, as well as their encoding as configuration data for such computers, reflect the safety principles of the system. This leads to a number of requirements set by standard bodies such as the European Union Agency for Railways [21], infrastructure managers such as Network Rail [71], and equipment suppliers such as [redacted] based on past project experience [84]. The existing process of verifying scheme plans against these constraints is

a manual and iterative one. With a typical scheme plan containing hundreds of instances (application points) for each design constraint, this process is prone to human errors caused by fatigue in performing repetitive tasks.

At ██████████ a toolchain for automating scheme plan verification has been developed based on Satisfiability Modulo Theories (SMT) solving using the Z3 prover. This toolchain, shown in Figure 1.2, takes a scheme plan and a so-called *design rule* that represents a static property of the scheme plan — encoded in DSLs that correspond to the *Track-Distance Model (TDM)* — and transforms them into a directed graph representation (the *Node-Edge Model* or *NEM*). This directed graph is then encoded into SMTLib2, the language of the Z3 prover, for automatic verification. In case a violation of the design rule is found, the toolchain gives a textual description containing identifiers of elements involved in the relevant *subproperty* (instance of design rule).

1.1 Our Contributions

Our work aims to streamline the scheme plan verification workflow, in particular to reduce redesign time following early discovery of errors in large scheme plans with many subproperties. We do this by implementing a graphical and (enriched) textual representation of design rule violations. This involves:

- A focus group study with users of the toolchain to establish the use case and requirements on the presentation of counterexamples.
- Development of a software architecture covering the components, interfaces and communications required to facilitate counterexample visualisation. This consists of:
 - Extracting counterexample information from the proof tool.
 - Defining an extended DSL for visualisation data.
 - Transforming the extracted information into the extended DSL.
 - Extending a domain-specific data authoring program with the ability to display visual representation of counterexamples using information in the extended DSL.
- Implementation of the functionality following this architecture, through both C# programming and extensions to the data formats involved, demonstrating that our concept works.

In this thesis, we present a working prototype developed within the industry environment of ██████████ engineering software tools, which demonstrates that unsat cores (counterexamples produced by Z3) can be enriched with enough data to visualise counterexamples in a graphical editor for scheme plan data.

1.2 Presentations and Publications

Our work in this project has resulted in the following presentations and publication:

- Presentation at the 38th British Colloquium for Theoretical Computer Science, 11–13 April 2022 at Swansea University, UK [74].
- Talk and abstract at the 26th International Workshop on Algebraic Development Techniques, 28–30 June 2022 in Aveiro, Portugal [9]
- Our focus group study for establishing requirements on counterexample visualisation was incorporated into a paper presented at the International Conference on Reliability, Safety, and Security of Railway Systems, 10–12 October in Berlin, Germany [3].

1.3 Overview of the Thesis

The rest of the thesis is organized as follows: Chapter 2 looks at some prior work in the fields of domain-specific visualisation and railway infrastructure verification; Chapter 3 introduces railway concepts used in the thesis, including the data model used by [redacted] engineering tools to represent the railway and the use of design rules to express static properties; Chapter 4 uses an example to introduce SMT solving with Z3, and its specific use within the verification toolchain; Chapter 5 describes the data formats, programs and libraries involved in our software development; Chapter 6 discusses the focus group study we ran to develop the requirements for counterexample visualisation designs; Chapter 7 describes the overall software architecture of our extended toolchain, and some design principles we follow throughout the development; chapters 8 and 9 detail our modifications to two pieces of [redacted] software in the verification toolchain — the [redacted] [redacted] [redacted] and the [redacted] [redacted] [redacted] [redacted] ([redacted]), respectively; finally, Chapter 10 presents an extended end-to-end demonstration of the verification workflow after our modifications to enable counterexample visualisation.

A Note on Railway Terminology In this thesis, we use a significant amount of railway industry jargon. Terms specific to the topic of this thesis are explained in Chapter 3 while other, more generic terms are listed in Appendix A. When a term of the latter category appears for the first time, it is marked *like this*^T.

Part I

Background

Chapter 2

Related Work

Contents

2.1	Domain-Specific Visualisation	7
2.2	Railway Infrastructure Verification	8

2.1 Domain-Specific Visualisation

There is a wealth of work done on visualising domain-specific models. A particular example of this is the B-method, a general-purpose formal method with good tool support for code generation [2]. The B-method has been used in model-driven development of safety-critical software, including software found in railway control systems.

Idani et al. [30] developed an approach to modelling railway topologies and signalling systems in three parts: first, the authors developed a graphical domain-specific language (DSL) based on the model-driven engineering (MDE) paradigm; second, they used the B-method to define the underlying operational semantics of the DSL; third, they used their Meeduse tool to automatically translate the DSL meta-model into the equivalent B specification, then translate the variable valuations returned by ProB [40] back to the DSL, in turn modifying the original business model. Our work similarly involves both graphical DSLs and formal methods, but we focus on transforming the counterexample generated by a proof tool back into the DSL, and presenting it as an overlay on top of the original model instead of modifying it. In addition, while this paper checks *dynamic* properties of the railway (how it behaves w.r.t. events), our work verifies *static* properties of the infrastructure (correctness of the topology in a ‘frozen’ state, without any events).

Researchers at Heinrich-Heine-Universität Düsseldorf developed tools to visualise the state of B models in a domain-specific style, illustrated using examples of a lift and various mathematical puzzles [38, 41]. The tool in [41] allows the user to connect a model with its visualisation using an animation function written in B. This made the behaviour of the models easier to understand, and allowed errors that wouldn’t be obvious otherwise to be spotted. While our work similarly involves domain-specific visualisation, we are additionally performing a transformation between data formats, because the model used by the proof tool

to represent track plans is different from the one rail engineers use. We describe this in more detail in Chapter 5.

Previous research at Swansea University include the OnTrack tool [33], an algorithm for displaying track layouts without geographic information based on simulated annealing, and key-frame visualisation of counterexample traces [73]. OnTrack is based around a custom DSL for *scheme plans*^T and a number of model transformations that enable generation of formal models for use with proof tools. Our work differs in that we are interfacing with an existing DSL and toolchain (██████ and ██████ both described in detail in Chapter 5) developed in the industrial context of railway safety cases and compliance with standards such as EN 50128 [10]. Whereas the key-frame visualisation in [73] is designed for dynamic property violations, we produce visualisations for static property violations in the railway topology data itself.

2.2 Railway Infrastructure Verification

Formal methods have been applied to verify both traditional and more modern signalling systems. For instance, the specification of the *European Train Control System*^T [75] and the *European Rail Traffic Management System*^T [7] have been verified. Static checking of *interlocking*^T *control tables*^T to complement model-checking approaches has also been investigated [26]. Similarly to Idani et al., our work involves both graphical DSLs and formal methods, but they check *dynamic* properties while our work verifies *static* properties of the infrastructure.

The OVADO tool can be used for data validation based on the B-method [1, 39]. For data description, the authors rely heavily on the B-method’s set theory and first-order logic. However, as they perform static validation, the B-method’s abstract machine notation plays less of a role. They model a scheme plan as a one-dimensional Cartesian coordinate system, in which balises, signals, points, track circuits and blocks are placed. They are modelling informal requirements such as “each zone must be connected”. The OVADO tool is capable of validating the properties we are interested in, however as discussed in Chapter 1, it would not fit into the ██████ ecosystem of verification tools.

Luteberget developed a tool suite named *Junction* [42], which features verification of infrastructure data for consistency and compliance with rules and regulations encoded in a knowledge base. Runtime for verification tasks is in the order of seconds. This is for a workflow in which one new element is placed into a scheme plan, and it is verified if the placement of this element complies with the rules. We took inspiration from this work for visualising counterexamples. The workflow to be supported at ██████ differs from the one supported by Junction: there is a need to verify a whole scheme plan after its design rather than to verify ‘on the fly’ when adding a new element. Again, our choice of technology is driven by the need to have a uniform formal methods framework: ██████ ██████ has started to build several tools around SMT solving.

Chapter 3

Railway Concepts

Contents

3.1	Infrastructure Elements	9
3.2	Design Rules	13

This chapter introduces railway infrastructure concepts that are involved in the rest of this thesis, as well as the so-called *design rules* used by [redacted] to specify constraints on how equipment is placed on the railway. These correspond with the ‘System S’ and ‘Property P’ in Figure 1.1 respectively. Note that we describe railway concepts in the context of the main rail network in Great Britain owned and maintained by Network Rail [69].

3.1 Infrastructure Elements

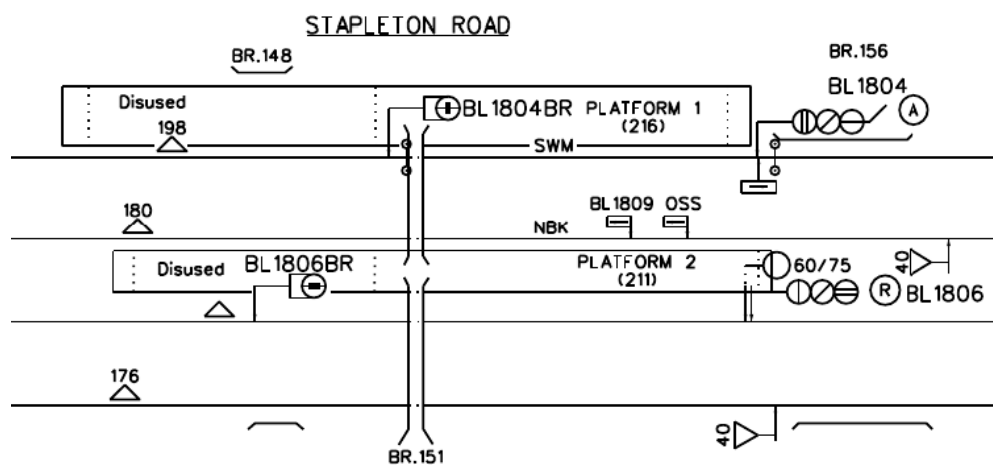


Figure 3.1: An excerpt of a scheme plan for a railway line near Bristol, courtesy of Network Rail.

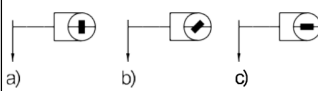
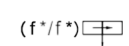
Multi-unit three-aspect signal: red/yellow/green. Post mounted.		03
Banner Repeater, three-state a) Normally "ON"; b) Normally "OFF"; ⁽¹⁾ c) Normally "OFF"; ⁽²⁾		68
TPWS, TSS Loops. (Arming and Trigger). "★" – Frequency. Normal direction frequencies not normally shown. Arrow indicates direction of travel.		74
AWS, non-standard distance. For example 230m for line speeds over 100mph.		63

Figure 3.2: An extract of the Network Rail standard on scheme plan symbols [66].

Railway signalling engineers use scheme plans to describe the track layout and control system behaviour (e.g. **routes**^T that can be set by an interlocking) for a piece of railway. Figure 3.1 shows an example of what may be found on a scheme plan. The meaning of all scheme plan symbols can be found in the relevant Network Rail standard [66]; Figure 3.2 shows an extract of this standard with the symbols involved in the example scheme plan we use in this chapter¹. Note that in this particular scheme plan, there is a deviation from the standard in that the arrow and vertical line through the TPWS TSS loop are omitted.

In this section, we will illustrate rail infrastructure concepts most relevant to our work using this example. For a more thorough introduction, the reader may want to refer to [25] and [77]. For a glossary of railway terminology, see also the glossary published by RSSB [78] and the Jargon Buster by Network Rail [68].

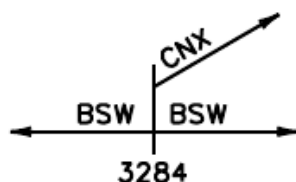


Figure 3.3: Change of ELR shown at a junction on the scheme plan.

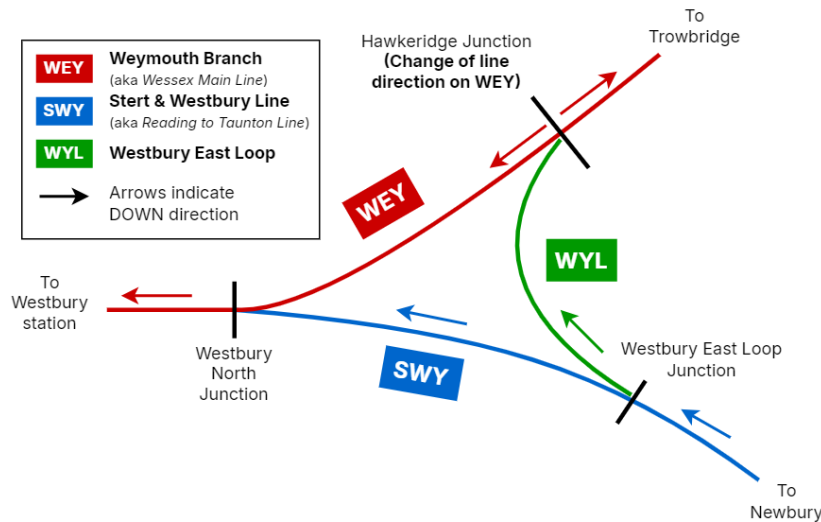


Figure 3.4: A simplified diagram of the triangle formed by the WEY, SWY, and WYL lines near Westbury station.



Figure 3.5: Signal BL1804, located at the end of platform 1 at Stapleton Road station.

3.1.1 The Track-Distance Model (TDM)

Each railway line in Great Britain is identified by an *Engineer's Line Reference (ELR)*^T which are three-letter codes, sometimes suffixed by a number². The ELR for the railway line in Figure 3.1 is **BSW** (as shown on Figure 3.3), reflecting its origin as the Bristol and South Wales Union Railway [72].

Every location along a given railway line has a unique distance (also known as *mileage*) measured from a *datum*^T, which is defined to have a distance of zero. The datum point for the BSW line is Bristol Temple Meads station [12]. This gives Stapleton Road station a distance of 1 mile and 50 chains, or 2.615 kilometres [70].

Directions on railway lines in Britain are typically named *Up*^T and *Down*^T, with the *Down* direction leading away from the line's datum and the *Up* direction leading to the datum. This leads to reversal of line directions at triangular layouts, such as the one shown in Figure 3.4.

¹Abbreviations: TPWS = Train Protection and Warning System; TSS = Train Stop Sensor; AWS = Automatic Warning System.

²The excellent *Railway Codes* website [12] offers a list of ELR codes.

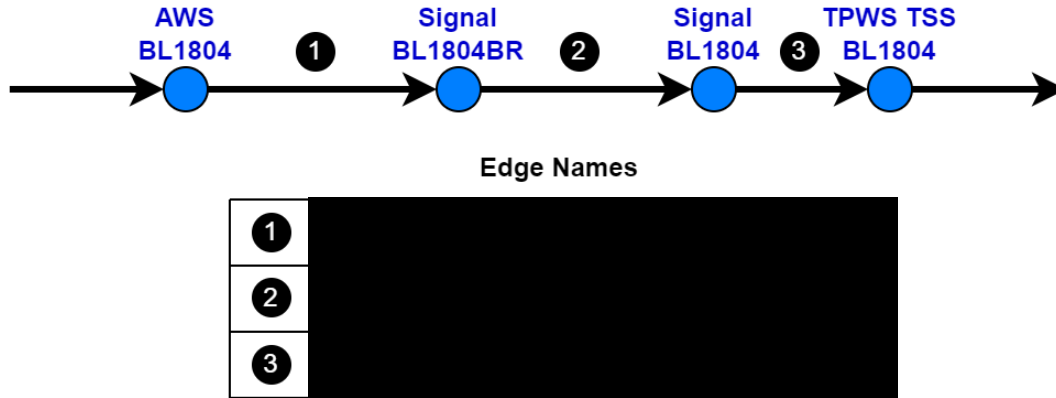


Figure 3.6: The UFR track at Stapleton Road represented in the Node-Edge Model.

Tracks on a multi-track railway line are each named after its direction and role: In Figure 3.1, the four tracks on the BSW line are, from top to bottom, Up Filton Relief (UFR), Down Filton Relief (DFR), Up Filton Main (UFM), and Down Filton Main (DFM).

The location of signal BL1804 (Figure 3.5) can therefore be expressed as

BSW UFR 2.716km

In the development of engineering software at [REDACTED] this model of the railway in which each location is referenced by a *track* (in this case **BSW UFR**) and *distance* (in this case **2.716km**) is called the *Track-Distance Model (TDM)*.

3.1.2 The Node-Edge Model (NEM)

While the TDM is an intuitive view of the railway for signalling engineers, it is more straightforward for proof tools to operate on scheme plan data as a directed graph — in our case, this allows for automated translation into SMTLib2, the input language of the Z3 solver described in Chapter 4 [3]. At [REDACTED] this directed graph representation of the railway is called the *Node-Edge Model (NEM)* [15]. In the NEM, each piece of equipment placed on a track is assigned a node, with the track becoming directed edges that connect the nodes. The procedure for transforming TDM infrastructure data into NEM is described in [82].

Using our example scheme plan, we can represent the equipment on the UFR track in NEM as shown in Figure 3.6. Note that the direction of each edge is reflected in its identifier: an edge starts at its node0 (shown in orange) and ends at its node1 (shown in pink).

3.1.3 ETCS Equipment

The European Train Control System (ETCS) is the signalling component of the European Rail Traffic Management System (ERTMS), which aims to replace the various national train control systems in Europe to create a seamless and interoperable European railway system [93]. As of 2022, the UK is in the process of introducing ETCS Level 2 on its network [92],

which provides the possibility of higher capacity and operating speeds thanks to its use of in-cab signalling (i.e. the **movement authority (MA)**^T is communicated to the driver through a display in the driving cab, rather than on lineside signals) and more precise train detection (which means the blocks can be shorter) [18].

ETCS Level 2 introduces two new types of physical equipment on the railway:

- **Balises:** Beacons placed in the middle of the track to act as reference points for the odometry equipment on the train. Some balises can also send other types of fixed messages known as *telegrams* [20] to the train.
- **Marker Boards:** These are signs placed on the side of the railway line to indicate where the driver should stop their train if the in-cab display indicates that the train is approaching the end of its MA.

The correct placement of these new pieces of equipment is critical to ensure that they can work safely with existing infrastructure. Furthermore, the position data of infrastructure elements in a scheme plan is used as input to control computers for ETCS, known as *Radio Block Centres* (RBCs). If these computers are programmed with incorrect scheme plan data, the safety of the railway could be jeopardised. Thus, quality control measures are needed to ensure the correctness of scheme plan data.

3.2 Design Rules

Layout constraints for railway infrastructure come from a number of sources, such as industry standards (one of them being the ERTMS standard [19]), requirements for a product (e.g. the [redacted] series of RBCs), and project-specific requirements. At [redacted] *design rules* are used to express design constraints for infrastructure elements such as balises. Each rule is written in a standard format, consisting of:

- An **identifier**.
- A natural language **description** of the constraint.
- A **rationale** explaining why specific values are chosen.
- A boolean **safety criticality** flag, with an accompanying textual description of the reasoning, and consequences of not complying with the design rule.

As an example, the following design rule *BG-05* concerns the spacing between *balise groups* (BGs):

Example 1: BG-05 Design Rule

Description Designed minimum spacing between adjacent balise groups shall be constrained by: $\geq \text{MIN_BG_SEPARATION}$ between adjacent end balises, one at each end of the two groups.

Rationale This is a design decision based on the maximum [sic] spacing permitted between adjacent balises of the same BG (Subset-040 4.1.1.2).

Safety Criticality Not considered directly safety critical. Non-compliance could lead to a failed balise read. Can be considered safety critical for a particular BG if that BG is required to be read for a safety critical purpose.

Note that in this case, the value of MIN_BG_SEPARATION is defined according to the specific needs of a project.

At the time of writing, the following design rules concerning balise groups (alongside rule BG-05 shown above) were supported by the [REDACTED] (Section 5.2.1). We now present the verbatim description of each rule taken from [84].

BG-01 Designed spacing between balises and AWS or TPWS transponders shall be constrained by:

1. $\geq$ [REDACTED] m between balise and end of AWS ramp.
2. $\geq$ [REDACTED] m between balise and edge of TPWS loop.

BG-02 Designed spacing between balises and train detection section ends shall be constrained by:

1. $\geq$ [REDACTED] m (or [REDACTED] – conditions apply) between balise and section axle counter head.
2. $\geq$ [REDACTED] m (or [REDACTED] – conditions apply) between balise and section insulated joint.
3. $\geq$ [REDACTED] m between balise and edge of tuned zone with tuned zone clear of balises.

BG-03 Design placement of balises should avoid *points*^T and *crossings*^T. Designed spacing shall be constrained by:

1. $\geq$ [REDACTED] m between balise and point *toes*^T.
2. $\geq$ [REDACTED] m between balise and point *frog*^T.
3. $\geq$ [REDACTED] m lateral separation between a balise on one path and the centre line of the other path.
4. No balises between the toe and frog of set of points.

Thesis Note: The original design rule document includes the diagram in Figure 3.7 to illustrate the requirements of rule BG-03 in relation to a set of points.

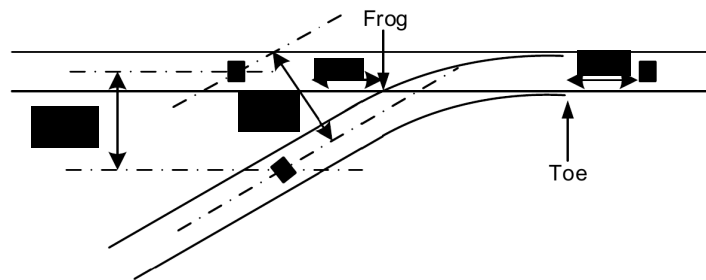


Figure 3.7: Diagram from [84] illustrating the requirements of design rule BG-03.

3.2.1 Applying Design Rules to a Scheme Plan

There are about 300 rules in total, 60 of which concern the placement of balises. In addition, scheme plans in real life can be quite large — a typical plan consists of more than 2000 pieces of equipment on 200 tracks [86]. Often, a rule can have many instances within a scheme plan — for example, the above rule about the spacing between balise groups can apply to multiple pairs of end balises.

As a result, checking scheme plans manually can be quite cumbersome. To cope with the increasing workload as the UK installs ERTMS/ETCS on more railway lines, the rail engineers need to automate the checking process.

Chapter 4

Underlying Proof Technology

Contents		
4.1	Satisfiability Modulo Theories	18
4.2	Z3 and SMTLib2: An Introduction by Example	19
4.3	Workflow of the [REDACTED]	22

The [REDACTED] [REDACTED] [REDACTED] uses the Z3 prover [63] to verify scheme plans against constraints. This chapter explores the concept of SMT solving, the working of Z3 and its language SMTLib2, and the algorithm used by the [REDACTED] for verification.

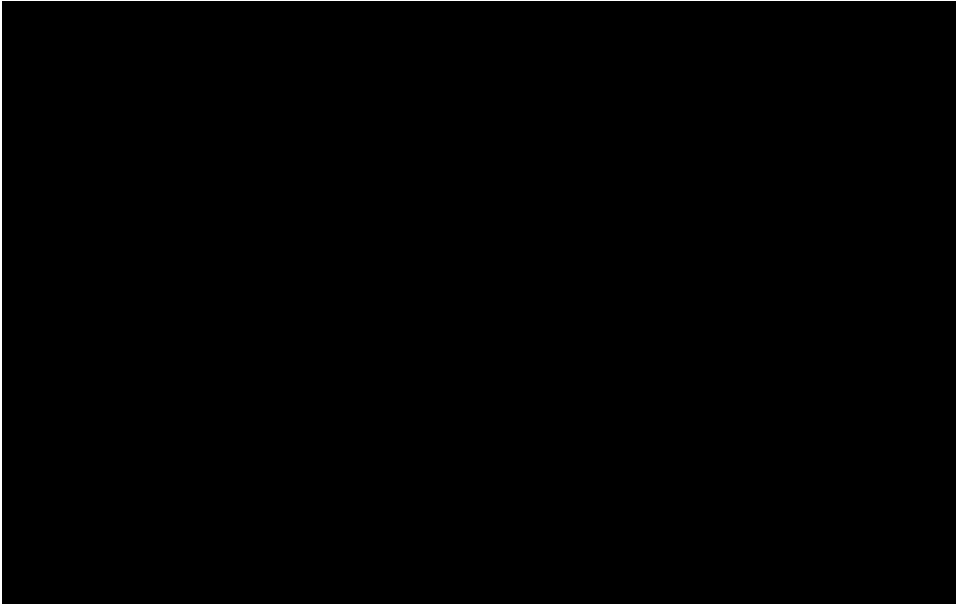


Figure 4.1: An overview of the [REDACTED]'s workflow.

This chapter explains the process illustrated on the right-hand side of Figure 1.1, and in more detail in Figure 4.1. We now provide a brief introduction to some terms in propositional

logic used in this chapter, following the more thorough explanations in [79].

Propositional Variable A symbol that can have the value *true* or *false*. In the text that follows, we use the letter p to denote a propositional variable.

Signature A finite set of propositional variables.

Formula Given a signature Σ , a formula in propositional logic is defined as:

- All $p \in \Sigma$ is a formula.
- $\perp$ is a formula.
- If φ and ψ are formulae, then $\varphi \Rightarrow \psi$ is a formula.

Operators such as $\wedge$ and $\neg$ serve as “syntax sugar” to shorten formulae that can otherwise be written using only these definitions. In the text that follows, we use the letters φ and ψ to denote propositional logic formulae.

Model An interpretation of symbols in a signature, i.e. an assignment of each variable to *true* or *false*. In the text that follows, we use the letter M to denote a model. $M(p) = \text{true}$ means “ M assigns p to the value *true*”.

Satisfaction Relation Given a model M , the satisfaction relation $\models$ is defined as:

- $M \models p$ (read “ M satisfies p ”) if and only if $M(p) = \text{true}$.
- $M \not\models \perp$.
- $M \models (\varphi \Rightarrow \psi)$ if and only if $M \models \varphi$ implies $M \models \psi$.

Satisfiability and Validity Given a formula φ , we say φ is *satisfiable* if there exists a model M such that $M \models \varphi$; we say φ is (*universally*) *valid* if for all models M , $M \models \varphi$. It follows that

- $\neg\varphi$ is satisfiable if and only if φ is not universally valid.
- Conversely, φ is universally valid if and only if $\neg\varphi$ is unsatisfiable.

4.1 Satisfiability Modulo Theories

Applying formal methods to software engineering often requires determining satisfiability of formulae in more expressive logics such as first-order logic [4]. Although satisfiability in first-order logic is only semi-decidable, the application usually concerns a decidable fragment of first-order logic, restricting interpretation of symbols to be consistent with a background *theory* — for example, the theory of Presburger arithmetic [90] restricts the meaning of the “+”

symbol. Applying such restrictions can turn the semi-decidable general satisfiability problem into a decidable one in the context of the theory, which allows it to be solved using automatic proof tools.

Satisfiability Modulo Theories (SMT) is a class of problems in the form

Is φ satisfiable in the background theory T ?
i.e. Is there a model of T that makes φ true? [5]

Solving SMT problems also lets us solve problems known as *Validity Modulo Theories*, which ask if a formula φ is satisfied by all models of a theory T . This is possible because φ is valid in T if and only if $\neg\varphi$ is unsatisfiable in T [79].

Software systems that solve SMT problems are called *SMT solvers*. These are often used to verify programs, by encoding their logic as logical formulae.

The SMT-LIB standard [5] specifies the language, semantics, logic and background theories used by SMT solvers. One solver that uses this format (version 2, known as SMTLib2) is Z3, developed by Microsoft Research. The underlying logic of SMTLib2 is a version of many-sorted first-order logic with equality. Z3 supports many of the theories specified in the SMT-LIB standard, such as arrays, bit-vectors, and linear real arithmetic.

SMT solvers check the satisfiability of a set of formulae by connecting each formula with $\wedge$ into an overall formula. When a solver determines that a set of formulae is unsatisfiable, we can ask it for an *unsatisfiable core* (or *unsat core* for short), which is minimal subset of the overall conjunction that is itself unsatisfiable [5]. For example, the unsat core of the propositional logic formula $(A \wedge B) \wedge (A \wedge \neg A)$ is $(A \wedge \neg A)$.

4.2 Z3 and SMTLib2: An Introduction by Example

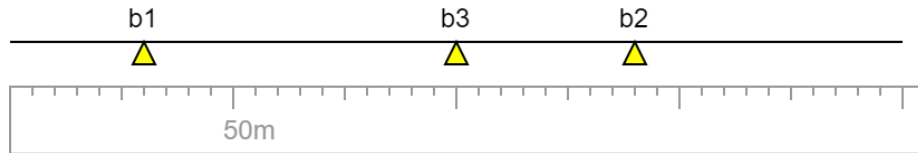


Figure 4.2: The scheme plan used in the SMT solving example.

We now illustrate the process of SMT solving with Z3 using an example. In this example, we encode the positions of the three balises in Figure 4.2 as integers, and check that they are all at least 50m apart from each other. This is the case for the pairs (b_1, b_3) and (b_1, b_2) , but not for the pair (b_3, b_2) .

Note that unlike typical Z3 tutorials such as [64], we focus on illustrating the process of checking scheme plan data against design rules as used by the XXXXXXXXXX rather than SMT solving in general. This informs our choice of constructs — for example, when defining an integer symbol named `number`, we use

```
(declare-const number Int)
(assert (! (= number 123) :named def_number ))
```

instead of

```
(define-fun number () Int 123)
```

because the former gives the definition a label and allows it to be included in an unsat core.

4.2.1 The Problem on a High Level

The question we are asking, expressed in natural language, is

Is the distance between every pair of balises at least 50m?

In this scheme plan, $Balises := \{b_1, b_2, b_3\}$. The property we want to check can be expressed as the formula (where integers represent distance in metres):

$$\forall_{x,y \in Balises} x \neq y \Rightarrow \text{distance}(x, y) \geq 50$$

Since every first-order logic formula is logically equivalent to a formula in *prenex normal form* [28], design rules to be checked by the [REDACTED] are expressed in this form to have a uniform structure. The nature of design rules for balises (which apply to *all* balises in a scheme plan) means these logic formulae start with a universal quantifier. To check such a design rule formula, we first instantiate it for every application point into a so-called *subproperty*:

$$\begin{aligned} & \forall_{x,y \in Balises} x \neq y \Rightarrow \text{distance}(x, y) \geq 50 \\ \Leftrightarrow & \forall_{x,y \in \{b_1, b_2, b_3\}} x \neq y \Rightarrow \text{distance}(x, y) \geq 50 \\ \Leftrightarrow & (b_1 \neq b_1 \Rightarrow \text{distance}(b_1, b_1) \geq 50) \wedge (b_1 \neq b_2 \Rightarrow \text{distance}(b_1, b_2) \geq 50) \\ & \wedge (b_1 \neq b_3 \Rightarrow \text{distance}(b_1, b_3) \geq 50) \wedge (b_2 \neq b_2 \Rightarrow \text{distance}(b_2, b_2) \geq 50) \\ & \wedge (b_2 \neq b_3 \Rightarrow \text{distance}(b_2, b_3) \geq 50) \wedge (b_3 \neq b_3 \Rightarrow \text{distance}(b_3, b_3) \geq 50) \\ \Leftrightarrow & \text{distance}(b_1, b_2) \geq 50 \wedge \text{distance}(b_2, b_3) \geq 50 \wedge \text{distance}(b_1, b_3) \geq 50 \end{aligned}$$

The final form of the above formula is the conjunction between three subproperties of the generic property applied to this particular scheme plan. This allows us to later check subproperties individually with Z3, i.e. we start each check with just the scheme plan definitions, then “overlay” one subproperty at a time so that we can identify the violating instance of the design rule.

4.2.2 Encoding the Problem as Input for Z3

To start expressing our problem in the SMTLib2 format, we declare three symbols in the theory Ints [5] that represent where the three balises are:

```
(declare-const b1_offset Int)
(declare-const b2_offset Int)
(declare-const b3_offset Int)
```

Next, we fill in the values by using the `(assert (! (...) :named ...))` construct. This adds an assertion onto Z3's *assertion stack* [5] and gives it a label (specified after `:named`), which lets Z3 include the assertion in any counterexamples [5].

```
(assert (! (= b1_offset 30) :named def_b1 ))
(assert (! (= b2_offset 140) :named def_b2 ))
(assert (! (= b3_offset 100) :named def_b3 ))
```

On an abstract level, we can think of each definition above as a propositional variable, and the assertion stack as a conjunction of such variables:

$$\varphi := \text{def_b1} \wedge \text{def_b2} \wedge \text{def_b3}$$

To verify the property we described in Section 4.2.1, we check each *subproperty* individually. As an example, the subproperty $\text{distance}(b_1, b_2) \geq 50$ can be expressed in SMTLib2 as:

```
(>= (abs (- b1_offset b2_offset)) 50)
```

The `(push)` SMTLib2 command adds an empty level onto the assertion stack; conversely, the `(pop)` command removes the most recent level from the assertion stack [5]. Using these commands, we can create a *scope* for each property in which we add the subproperty to the assertion stack, check the satisfiability of the conjunction of all formulae on the stack, then remove the subproperty to reuse the previous definitions for checking the next subproperty.

```
(push)                ; Add a level to the assertion stack
(echo "Checking b1 and b2")
(assert (!
  (>= (abs (- b1_offset b2_offset)) 50)
  :named subprop
))
; Put subprop on the top level
(check-sat)           ; Check satisfiability of formulae on the stack
(pop)                 ; Remove the top level
```

We can think of this as checking the satisfiability of the aforementioned conjunction with the subproperty appended as a propositional variable *subprop*, i.e. the formula being checked becomes

$$\varphi' := \text{def_b1} \wedge \text{def_b2} \wedge \text{def_b3} \wedge \text{subprop}$$

4.2.3 Checking with Z3

The `(check-sat)` command asks Z3 to search for a model M such that $M \models \varphi'$. There are three possible outputs for this command [5]:

- `sat`, if Z3 could find an M , i.e. φ' is satisfiable. In this case, we can issue further commands to obtain the model M [5].

4. Underlying Proof Technology

- `unsat`, if Z3 could prove that there is no M , i.e. φ' is unsatisfiable. In this case, we can issue further commands to inspect the proof [5] or the `unsat` core.
- `unknown`, if the search was inconclusive, i.e. Z3 can neither find an M nor prove the non-existence of M . This can be caused by, for example, a partial function in φ' if the background theory includes values of the function's input data type that are outside the domain of that function.

When we run Z3 on the source file of this example (complete source code in Listing C.1), we get the following output:

```
Checking b1 and b2
sat
Checking b1 and b3
sat
Checking b2 and b3
unsat
```

This shows that the first two subproperties have passed the verification, but the third subproperty does not. We can use the `(get-unsat-core)` command to obtain an *unsat core* for the violating subproperty (b_2 and b_3). In this case, the `unsat` core is:

```
(def_b2 def_b3 subprop)
```

This means the fragment `def_b2  $\wedge$  def_b3  $\wedge$  subprop` of φ' is an unsatisfiable formula. Assuming we encoded and instantiated the property correctly, we can interpret this as “the placement of b_2 and b_3 does not comply with the balise spacing requirement”. To make the scheme plan comply with the property, we then need to change the location of either b_2 or b_3 .

4.3 Workflow of the XXXXXXXXXX

The XXXXXXXXXX uses Z3 in a way such that the `check-sat` command cannot output `sat`, i.e. the only possible outcomes are `unsat` or `unknown` [86]. We can think of the XXXXXXXXXX's interaction with Z3 as the following function:

Function `unsat( $\psi$ )`

Input: Formula ψ

Output:

$$\begin{cases} \text{unsat} & \text{if we can prove that there is no model } M \text{ such that } M \models \psi \\ \text{unknown} & \text{if } \psi \text{ is satisfiable or Z3 fails to prove that } \psi \text{ is unsatisfiable} \end{cases}$$

This function is used in the overall algorithm of the XXXXXXXXXX:

Algorithm 1: Checking a scheme plan against a property**Input:** scheme plan θ , Property φ **Output:**

“Pass”	if $\theta \models \varphi$ can be proven with given resources
“Fail”	if $\theta \not\models \varphi$ can be proven with given resources
“Manual check required”	if neither $\theta \models \varphi$ nor $\theta \not\models \varphi$ can be proven with given resources

```

switch unsat( $\theta \wedge \varphi$ ) do
  case unsat do write “Fail”;                                /* Pass 1 */
  case unknown do                                           /* Pass 2 */
    if unsat( $\neg(\theta \wedge \varphi)$ ) = unsat then
      // ( $\theta \wedge \varphi$ ) is a tautology
      write “Pass”
    else
      write “Manual check required”
    end
  end
end

```

Given a scheme plan and a *design rule* that contains the formal encoding of a property φ , the XXXXXXXXXX encodes the scheme plan as a set of formulae θ in the SMTLib2 language, instantiates φ into a *subproperty* φ_i for each application point i on the plan, then performs verification in two passes:

- Pass 1:** Check if each conjunction $\theta \wedge \varphi_i$ in the set $\{(\theta \wedge \varphi_1), (\theta \wedge \varphi_2), \dots\}$ is unsatisfiable. If Z3 comes back with *unsat* for any subproperty, we know that the scheme plan violates the property, and can construct a counterexample based on the *unsat* core.
- Pass 2:** If Z3 gives us *unknown* for all subproperties in the first pass, we check if the negation of each subproperty $(\theta \wedge \neg\varphi_i)$ in the set $\{(\theta \wedge \neg\varphi_1), (\theta \wedge \neg\varphi_2), \dots\}$ is unsatisfiable. Recall from the beginning of this chapter that the negation of a unsatisfiable formula is universally valid [79]. This means if we can obtain *unsat* for all subproperties in this pass, then the scheme plan is compliant with the design rule.

If we cannot get either of the two outcomes mentioned above, we cannot establish if the scheme plan complies with the design rule with given resources. This situation can occur because we work with a semi-decidable theory (directed graphs) and the checks are bounded for performance reasons — for example, a search bound k is used when checking rules involving the “adjacent” relationship between a balise and another element, limiting the number edges traversed to k while looking for the other element, i.e. the XXXXXXXXXX’s adjacency calculation is a partial function.

The XXXXXXXXXX’s purpose is to find errors and prove correctness of scheme plans in the *decidable fragment* of the theory of directed graphs, and in practice a value of k is chosen

such that the time taken by the ████████ to verify a scheme plan will represent a significant improvement over manually performing the same check. Details about its performance optimizations and their implications on the interpretation of verification results can be found in [3]. The correctness of the ████████'s encoding of scheme plans and its conclusions from verification are beyond the scope of this thesis; rather, our focus is to provide enriched information to the tool's user in the case of a "Fail" result, based on the information we can obtain from the ████████ and Z3.

4.3.1 Applying Pass 2 to Our Example

Following the example in Section 4.2, a corrected scheme plan would give unknown from pass 1 of algorithm 1 for all three subproperties of the overall property described in Section 4.2.1. To check if the plan is valid, we check the satisfiability of the negation of each subproperty *subprop*:

$$\text{def_b1} \wedge \text{def_b2} \wedge \text{def_b3} \wedge \neg \text{subprop}$$

For example, to check the negation of the first subproperty involving b_1 and b_2 , we write the following in SMTLib2:

```

11 (push)
12 (echo "Checking b1 and b2")
13 (assert (!
14   (not
15     (>= (abs (- b1_offset b2_offset)) 50)
16   )
17   :named negated_subprop
18 ))
19 (check-sat)
20 (pop)

```

A scheme plan that complies with the rule would return *unsat* for each negated subproperty:

```

Checking b1 and b2
unsat
Checking b1 and b3
unsat
Checking b2 and b3
unsat

```

The full source code for the file used to perform pass 2 of the algorithm on the corrected scheme plan can be found in Listing C.3.

Chapter 5

Software Tools and Technologies

Contents

5.1	Data Formats	25
5.2	Programs and Libraries	30

5.1 Data Formats

The process of verifying scheme plan data against layout constraints uses different data models and formats for different purposes. An understanding of these and their purposes is necessary before we delve into the details of the process.

*This section has been removed for
intellectual property reasons.*

This section has been removed for intellectual property reasons.

This section has been removed for intellectual property reasons.

5.1.2 Formalizing and Encoding Design Rules

In order to use design rules for verification using software tools, we need to first capture the requirements in an unambiguous mathematical notation. At [REDACTED], the chosen notation is many-sorted first order logic, and the process of encoding these requirements is known as “formalization”, discussed in detail in [3].

The BG-05 design rule shown in Example 1 is formalized into the following first order logic formula:

$$\begin{aligned} \forall b, b' \in \text{Set}_{\text{Balise}} \\ & \text{adjacent}(b, b') \\ & \Rightarrow \text{balise_group_id}(b) \neq \text{balise_group_id}(b') \\ & \Rightarrow \text{distance}(b, b') \geq \text{MIN_BG_SEPARATION} \end{aligned}$$

This formula expresses that, the distance between two adjacent balises (as defined in the $\text{adjacent}(b, b')$ relation), each belonging to a distinct balise group, shall be greater than or equal to the constant **MIN_BG_SEPARATION**.

The design rule is then encoded into an XML file for use by software tools, containing both its original natural-language components and the logic formula. The XML file representing the BG-05 design rule can be found in Listing B.1.

5.1.3 REST and OpenAPI

Representational State Transfer (REST) [22] is a software architectural style used to build many web applications. The REST constraints [17] include the use of a client-server design pattern, and communicating using a uniform *representation* of a resource rather than the resource itself – for example, using JSON [14] to represent data structures.

One such application is the [REDACTED] whose functionality will be discussed in more detail in Section 5.2.1. The REST API used between its server and client is defined in an OpenAPI schema (YAML file) [88]. Listing 5.3 shows the definition of the *Report* data structure used to represent verification results.

Listing 5.3: Fragment of the [REDACTED]’s OpenAPI schema showing the structure of a report.

```

1 openapi: 3.0.0
2 # ...
3 paths:
4   # ... Definition of API endpoints
5 components:
6   schemas:
7     # ... Other data structures
8     Report:
9       required:
10        - id
11       type: object

```

```

12     properties:
13         id:
14             type: string
15             description: The identifier of the report.
16         dataid:
17             type: string
18             description: The identifier of the data, possibly a git commit ID.
19         jobid:
20             type: string
21             description: The identifier of the associated job.
22         startdatetime:
23             type: string
24             description: The time and data when the checking started.
25         checkingtime:
26             type: string
27             description: The total time to check the data.
28         gitparameters:
29             $ref: '#/components/schemas/GitParams'
30             description: The git parameters.
31         designruleresults:
32             type: array
33             description: A list of design rule results
34             items:
35                 $ref: '#/components/schemas/DesignRuleResult'
36         logs:
37             type: array
38             items:
39                 $ref: '#/components/schemas/LogEntry'
40             description: List of Logs.

```

The schema can be used to generate “model” classes in a chosen programming language that represent its data types, with a tool called Swagger Codegen [89]. These “model” classes can then be shared by the client and server components to abstract away the lower-level details of the interaction between them (e.g. HTTP requests and the raw content of a JSON response) — as long as the classes on the client and server were generated from the same OpenAPI schema, they can interact with data structures in the API as if they were native objects in the chosen programming language.

One benefit of these generated “model” classes is that serialization of data becomes easy. Listing B.2 shows an example report from the ████████’s API as specified in Listing 5.3. This report can be written to an XML file using the `System.Xml.Serialization.XmlSerializer` class from the .NET standard library [62]. The result of this serialization is shown in Listing B.3.

5.2 Programs and Libraries

Using the data formats introduced above, the following programs and libraries are involved in verifying scheme plan data. Here we provide a brief introduction to them — later chapters will explore them in more detail.

5.2.1 [redacted] Software

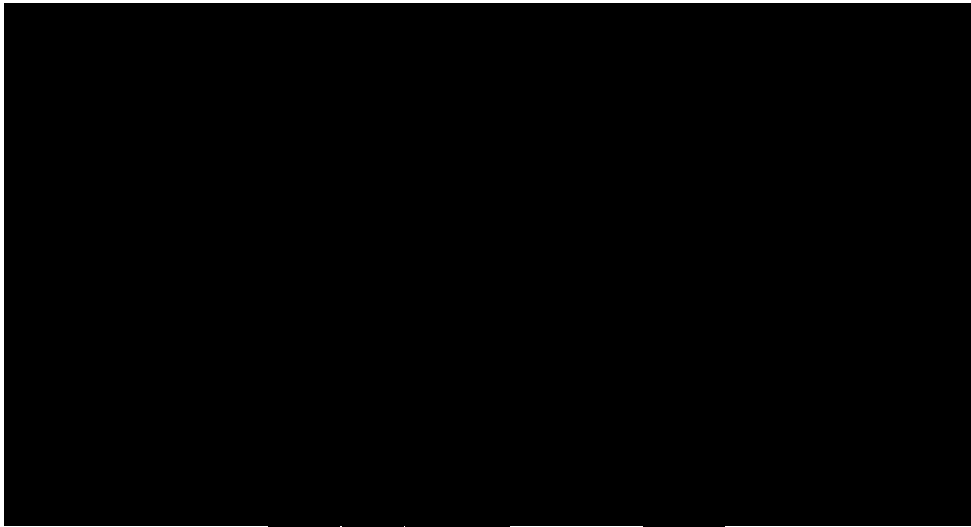


Figure 5.2: The “[redacted] [redacted] [redacted]” tab of the [redacted] user interface.

Table 5.1: Data formats used in communication between the tools

From	To [redacted]	To [redacted] client	To [redacted] server
[redacted]	-	Command-line arguments	-
[redacted] client	XML & [redacted] files	-	REST request
[redacted] server	-	REST response	-

The current workflow of a railway engineer wishing to check scheme plan data for errors involves the following pieces of software developed by [redacted]:

- The [redacted] [redacted] [redacted] [redacted]), a program that lets the user edit data in the [redacted] format, and interfaces with various other more specialised tools. It is capable of displaying data in a tabular form in the “Editor” tab, as well as graphically in the “Track Layout” tab. Our work in this thesis is based on version 3.0.14 of [redacted].
 - [redacted] uses a library developed by [redacted] called [redacted] for interacting with scheme plan data encoded in the *Track-Distance Model (TDM)* and stored in [redacted] files. Our work in this thesis is based on version 0.0.36 of [redacted].

- The **VerifTool** (which we will hereafter refer to as “**VerifTool**”), a tool that verifies scheme plan data against design rules. The **VerifTool** has a client-server architecture, in which the server and client communicate with each other over a REST API:
 - The **server** is designed to run in the cloud. It receives TDM data and design rule(s) to be checked from the client, converts the data to the *Node-Edge Model (NEM)*, then encodes both the data and the design rule as an SMT problem to be checked by Z3. When queried by the client, it can return the progress of a verification job and (if the job is finished) its outcome.
 - The **client** is a command-line program that runs on the engineer’s computer alongside **VerifTool**. The end user does not directly interact with the **VerifTool** client – instead, **VerifTool** issues shell commands when the user interacts with its “**VerifTool**” tab (shown in Figure 5.2), and monitors the client’s output to display information back to the user.

Our work in this thesis is based on version 1.0.9 of the **VerifTool**.

Table 5.1 shows the data formats used by the tools to communicate with each other.

5.2.2 Publicly Available Software

In addition to Z3 and SMTLib (introduced in Chapter 4) used by the **VerifTool** the **VerifTool** software described above uses the following publicly available technologies:

.NET and C# All **VerifTool** software introduced in Section 5.2.1 is written in version 10 of the C# programming language [59] and target the .NET 6.0 platform [52], both developed by Microsoft.

ASP.NET Core A framework developed by Microsoft for building web applications that run on .NET [54], used by the **VerifTool** server.

Windows Presentation Foundation A GUI framework developed by Microsoft [60] used by **VerifTool**.

CommandLineParser A C# library for defining and parsing command line arguments [80], used by the **VerifTool** client.

RestSharp An HTTP client library for .NET applications [65], used by the **VerifTool** client for its interactions with the REST API.

Docker While the **VerifTool** server is designed to run on its own dedicated server in production, during development we use Docker [13] to provide a reproducible and isolated environment for testing.

Part II

Contribution

Chapter 6

Designing the Visualisations: Developing Requirements

Contents

6.1	Establishing Requirements: Our Focus Group Study	35
6.2	Our Proposed Designs for Representing Single Errors	47

6.1 Establishing Requirements: Our Focus Group Study

The concept of a focus group dates back to the 1920s, originally as a market research technique. Nowadays, it has evolved into a data collection technique used across multiple disciplines. In healthcare, for example, it was used to explore how media messages affect people’s understanding of AIDS [35]. In software engineering, projects that involve human-computer interaction (HCI) often use it to gain an understanding of some aspect of user interaction design, such as privacy issues of photo sharing on social media sites [8].

In our project, since the software we produce will present design rule violations in a domain-specific graphical form, we felt it is best to consult the intended customers of our software in the target domain — namely railway engineers who work with ETCS infrastructure data — to come up with designs of graphical representations that are intuitive for them. The rest of this section describes how we conducted a focus group study in June 2022 to form the basis of the design of counterexample displays (“highlights”).

6.1.1 How a Focus Group is Typically Run

A focus group involves a small group of people who share similar characteristics, along with a number of moderators to lead the discussion. The participants are selected by researchers to provide ideas or feedback on the topic of the research. The “focus” of the discussion is a collective activity for the group, such as reviewing design prototypes or debate a set of questions [76].

The focus group session starts with an informal “icebreaker” conversation to relax the participants and enable them to engage in open, honest discussion. This is followed by a briefing that informs participants of the activities in the session, as well as their rights in the study. In particular, they need to be assured that their participation is entirely voluntary, and be informed about how the data collected in the study will be handled [37].

The majority of the time in the session is used for the group activity and discussion. How this is run varies by subject matter, but it is always planned in advance, guided by the moderators, and centred around material distributed to the participants beforehand. During this stage, the moderators are tasked with two main jobs:

1. Facilitating open discussion among the participants, so that they act as co-researchers through both complementary and argumentative interaction, in order to generate feedback that the researcher wouldn’t have had otherwise [36].
2. Keeping the discussion on topic, so that the time of the session is used effectively.

After the group activity, the moderators bring all the participants together for a plenary discussion. The aim of this stage is to come to an agreement on the conclusions of the session. In the case of designing user interfaces, this is to come up with a common design proposal. The session ends with everyone agreeing on an outcome.

After the session, the data collected (typically a recording of the discussion) is analysed to produce recommendations for the next steps in a research project.

Responsible Research and Innovation (required by EPSRC for UK universities) [95] requires that participation in a focus group is voluntary, and that all participants agree to any audio and/or video recordings of the event.

6.1.2 Preparation for the Focus Group

As the focus group study involved human participants, we obtained approval from the Faculty of Science and Engineering Ethics Committee at Swansea University, as required by university regulations. To this end, we presented a concept of the focus group to the committee.

In terms of content, we prepared a number of visualisations which we thought to be of good quality. The participants then should provide us with feedback in two forms: we wanted to know their preferred designs of the visualisation, as well as the reasoning behind the elements they choose to include. The visualisations we provided represented a common baseline on which the participants can suggest how best to highlight counterexamples. In hindsight, we could have had a pilot study to come up with realistic violation scenarios, but our contact with the engineers at [REDACTED] was quite limited at the time.

To this end, we first categorised the design rules based on common patterns, and came up with one example violation for each category; when presenting these violations (without highlights), we also made sure to use language and symbols that are already familiar to the railway engineers.

6.1.2.1 Classification of Design Rules

We studied a document provided by [84] containing design rules related to balise group (BG) placement, and looked for patterns in how the rules are formulated. In general, such design rules take one of the following forms:

1. **Distance constraints:** A balise must be at least / at most x metres from *<existing equipment>*.
2. **Minimum / maximum separation:** Certain types of balises shall be placed at least or at most x metres away from another balise.
3. **Prohibition of placement:** No balises shall be placed in a designated *area^T*.
4. **Requirement of placement:** A balise of type t shall be placed at a designated *position^T*.
5. **Balise group density:**
 - a) At most x balise groups shall be placed between a and b / within *<area>*; or
 - b) At least one balise / BG shall be placed within *<area>* / at *<position>* / between a and b .

The rules speak solely about *balises* and *balise groups* (BGs) as track equipment. The rules concern *distances* (“A balise must be at least / at most x metres from *<existing equipment>*”), *areas* (“No balises shall be placed in *<area>*.”) and *positions* (“A balise of type t shall be placed at *<position>*.”)

Consequently, we needed to invent highlighting mechanisms for balises and balise groups, in case they have been wrongly placed, and visualisations for distances, areas and positions. In order to explore these, we anticipated that the software we produce needs to have a generic visualisation for each of the five classes of design rule violations, with additional information (such as the required distance value or type of equipment) encoded as parameters to the generic visualisation. This also meant we needed to present one example violation of each class to the participants of the focus group.

6.1.2.2 Material We Provided

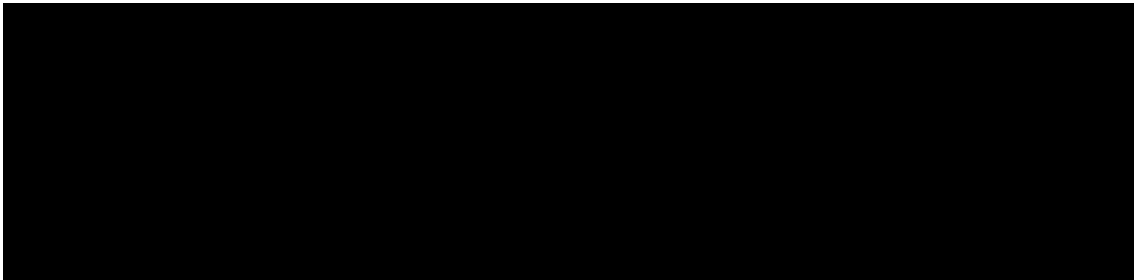


Figure 6.1: The scheme plan used in the focus group examples.

6. Designing the Visualisations: Developing Requirements

With the classification of design rules, we chose a design rule of each class, and based our example violations on a common scheme plan, shown in Figure 6.1. We presented the examples each in an exercise sheet, in the following fashion:

1. We quote the design rule in question verbatim.
2. We show a scheme plan diagram with exactly one violation of the design rule in question, where the participant can draw their design of the highlighting.
3. Finally, we present a suggested textual description of the error.

This is accompanied by instructions on the top, and room for comments at the bottom. One example of an exercise sheet is shown in Figure 6.2.

In addition to drawing their preferred visualisation, the participants were also asked to provide their preferred textual description, as well as suggestions for how the editor should present the error (in terms of the UI elements used, and how the presentation relates to other parts of the application).

Author: _____ Version: _____

Task 1

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.
Think about:
 - How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
3. Describe how you want the editor to display the textual description.

[BG-04] Designed maximum spacing between adjacent balise groups shall be constrained by: $\leq 1500\text{m}$ between reference balises.

Key to Symbols

Track T1

Track T2

Track T3

Balise

End Node

Simple Node

Points Node

Undirected Position

Suggested Textual Description The distance between reference balises b_1 and b_2 is larger than 1500m via track T1.

Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

Figure 6.2: Example of an exercise sheet used in the focus group study.

The full exercise sheets are shown in Appendix D.3.

6.1.2.3 Demographics of the Group

The participants of the focus group consisted of the following:

- 8 software engineers, working on the editor and [REDACTED] projects
- 3 systems engineers
- 2 technical authorities, who are experts in a particular subject
- A project manager, who monitors progress and manages risks related to projects
- A research manager, who oversees and organises the various research activities, as well as collaboration with universities (through the UKRRIN network [91]).

Of the participants, half ($n = 7$) had 20 years or more experience in the railway industry (in particular, the technical authorities had more than 30 years), while most of the rest (5) had less than 10 years of experience. 2 of the 15 participants were female (13%), which is close to the industry-wide average of 16%¹ [96].

6.1.2.4 How We Ran the Session

In line with the general structure of focus groups described in Section 6.1.1, we started by greeting the participants, followed by a briefing. During the briefing, we distributed an information sheet and a consent form (Section D.1) to each participant, and gave a presentation (Section D.2) informing them of what will happen during the session, and how we planned to use the information collected from them.

The participants were then evenly split into three groups of 5–6 participants each for the main group activity, which lasted approximately one hour. Each group was given copies of exercise sheets such that each participant has the ability to work through several drafts for each example violation. Within each group, the participants were free to decide how to work through the examples — some preferred to come up with individual proposals before comparing with the rest of the group, while others worked on a single set of proposals collectively. By the end of the group activity, we requested that each group chooses one set of designs (i.e. one design for each example violation) to be presented in the plenary session.

The plenary session lasted an hour. During this session, we went through the example violations. For each example, we let each group present their chosen visualisation design, then discussed how the groups' proposals compare to each other, and came up with a common proposal for the example in question before moving on to the next example. The proposed designs chosen by each of the groups are shown in Appendix D.4. The plenary discussion was recorded on video (with the participants' consent) for analysis after the focus group session.

The day finished with a short summary by the researchers of the conclusions from the session, and a description of the next stages of the project in relation to the user interaction design.

6.1.3 Findings of the Focus Group Study

While we received the feedback we asked for (visualisation designs), there were some surprising findings from the focus group — specifically, all three groups of participants highlighted

¹Note, however, that only 5% of the workforce in suppliers — the sector [REDACTED] is in — are female.

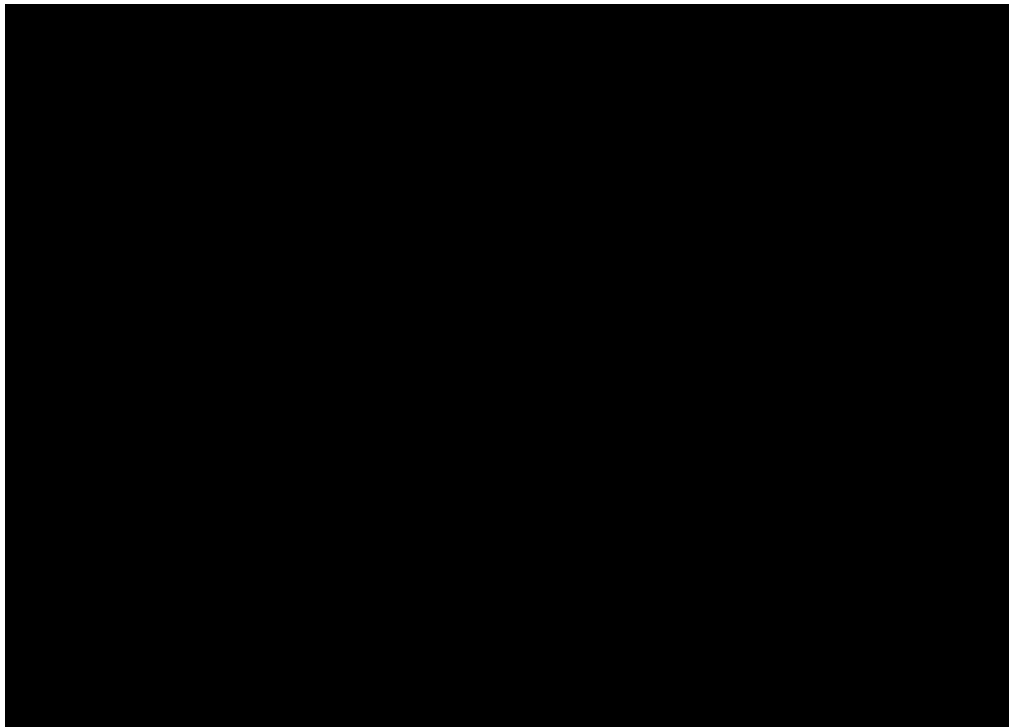


Figure 6.3: Design presented by group 1 for task 1.

the importance of fitting the visualisation into the workflow of the [REDACTED] tool. One group also decided to present the visualisation of one example violation as a 5-step process, instead of a single graphical design for each example. The majority of time in the plenary discussion was spent on workflow needs, whereas it was relatively easy to come to an agreement regarding the visualisation designs and textual descriptions.

To illustrate the discussion that occurred in the plenary session, we now show a selection of scanned completed exercise sheets from the focus group session, along with some quotes we have taken directly from the transcript of the video recording, presented in the order of the original discussion. The full transcript of the plenary discussion can be found in Appendix D.5.

The participants largely agreed that the visualisation should include both highlights on the scheme plan and a textual description with details of the error. However, there was some debate on what information is included. For example, group 1 described their design for task 1 (Figure 6.3) as follows:

We show the distance as a dotted line, show the actual distance value, and the expected value in brackets. The error message can pretty much stay as it is, but we have a short and long version, where the long version also includes the track number and the positions of balises

Group 2 (Figure 6.4) came up with largely the same design, but added a suggestion for displaying multiple errors without cluttering up the UI:

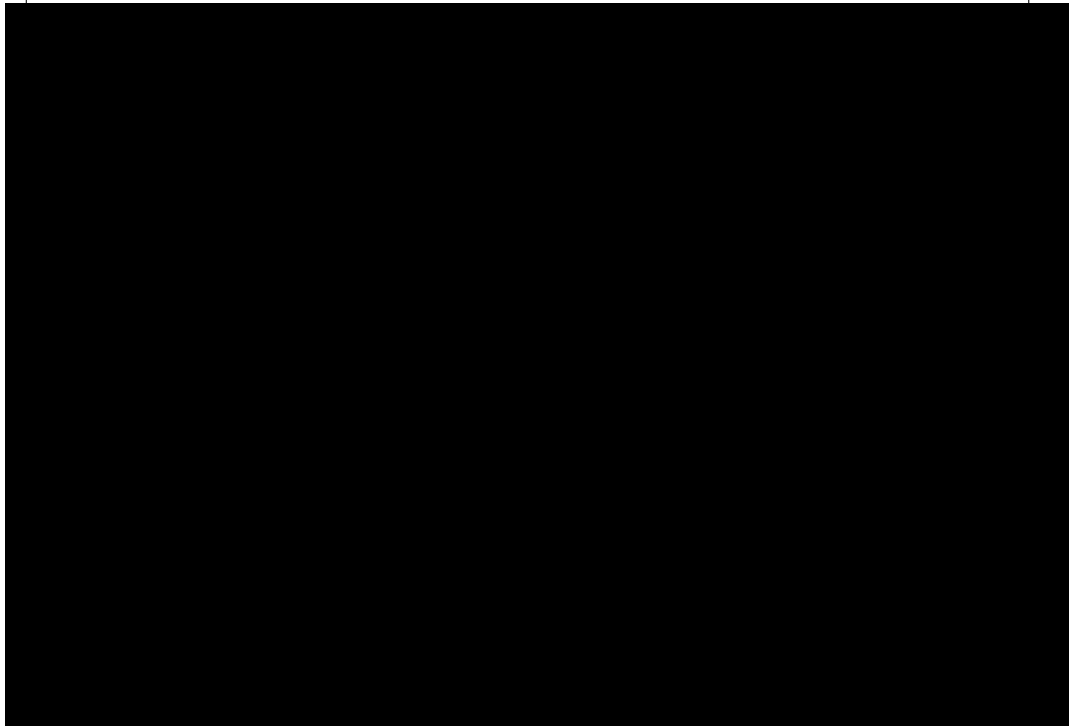


Figure 6.4: Design presented by group 2 for task 1.

We highlighted the objects that were referred to by the error, i.e. the balises and the path between them. We also said it's useful to know the actual distance — if it's only a couple of metres out, we don't care.

Then we got into the discussion of how to display it. We assume there will be some kind of clickable error list. The highlight only appears when you click on / hover over the error. When you click on it, you also get the actual rule (verbatim text) and the full path.

In contrast with the first two groups, group 3 proposed that we have a unified presentation of all errors, and instead focused on designing a sequence of interactions in 5 steps, starting with an overview of all errors on the plan and ending with the granular details on specific objects.

We took the idea that rather than having a different way of visualising each [type of] error, we have one way of visualising all the errors, with the caveat that we're looking at this from the perspective of a data entry person, at the end of the design workflow.

One of the steps they presented, shown in Figure 6.5, included showing a list of errors when an element is selected, and removing information not related to the selected element:

We show a drop-down list of all the errors [on an element]. Once you select the error that you want to address (of course, a bit of thinking is needed to figure out

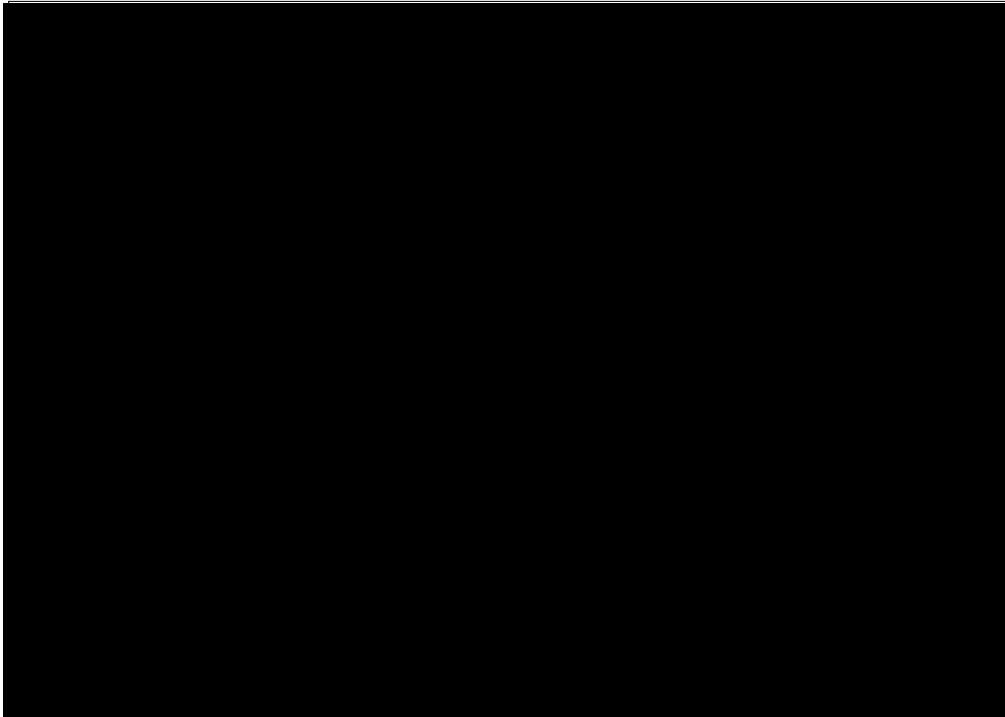


Figure 6.5: A step in the workflow sequence designed by group 3.

the priority), irrelevant information is removed — these can be quite busy scheme plans, and if I put a highlight over a section, there could be multiple objects in that section that don't matter for the problem at hand; but maybe they do, which is why we say we remove the irrelevant stuff ...

One engineer added that the user should also be able to subsequently re-display the “irrelevant” information:

We had that discussion. We thought that while automatic hiding is good, we also want the ability to bring them [the highlighting] back on again, because you need to have the whole picture.

There was then some debate on the contents of the textual descriptions, which revealed an underlying issue: not all design rules were created equal.

“There are two extremes: one is to just tell them what the error is; the other is to tell them what to do to fix the error. The danger with the latter is that the error may not actually be the problem. So how far back down that spectrum do you go? Do you just say that the distance is too large?” — Engineer 9

“It comes down to how they come up with the rules in the first place [...] What things have they considered in order to come up with this rule? To be able to come

up with how to fix it, you have to take all that into consideration, and then provide an answer.” — Engineer 14

“The other thing is, some of these rules are desirables, as opposed to must do it. Therefore, it’s an engineering compromise to decide which rules we have to respect, and which ones we’re not too bothered about in the constraints we’ve got. You’re never going to get a tool to make that sensible decision — it’s actually going to mislead the human by giving them the obvious one, which may often be the wrong one!” — Engineer 9

Later, one engineer pointed out that the rules themselves may need to be associated with additional priority attributes:

It’s not just in terms of the safety criticality, it’s the significance for needing redesign that we really want to prioritise on. You obviously can’t quite do that [right now], but yes you would look at things like repositioning balises after points, and where you put a balise in relation to an EoA, before you care about the maximum separation between balises.

In case the user decides that an error is not to be resolved, the user should be able to ignore that error for the time being:

“So step 6 is the “marker” where we can say “I’ve looked at this, but I’m not going to fix it, hide it from now on”. Here, we can either accept that it’s okay, or defer it until we’ve fixed the other problems.” — Engineer 4

“Precisely. At least hide it for now, because otherwise you won’t see the wood for the trees.” — Engineer 6

After the groups presented their designs for the first task, we had a discussion to sum up the points raised. During this discussion, participants brought our attention to the potential of adding clutter to already busy scheme plans:

For graphically showing the distance with the dotted line: it’s great but I think it will be difficult. When you look at a particular plan, depending on how it’s spaced out, it may not be easy to make sense of what is shown, because there’s so much other stuff. What we’ve seen when we try to display things is: trying to put labels on automatically makes it hard to be made readable. That’s why I’d go for the simpler approach of highlighting the path, then show the measurements (distances) in the text.

It was also suggested that we use different colours depending on the nature of the error, to differentiate between error classes involving similar objects:

We were also thinking: when we highlight it, we may want to use a different colour for a missing balise as opposed to a superfluous one. It’s a way to quickly see the nature of the error.

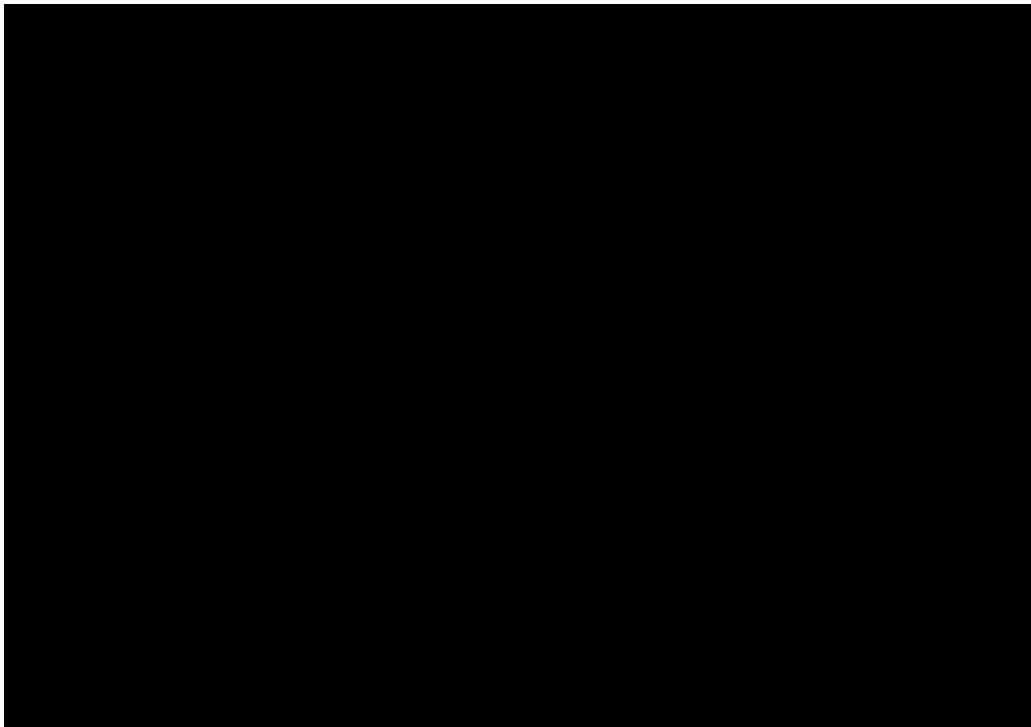


Figure 6.6: Group 1's response to task 3.

Later, the discussion moved onto the importance of context when presenting errors.

"It's important to put the problem into context. It could be highlighting an area, a route, or other elements involved in the error. When you read the information provided with the error, you should know what [the nature of] the error is." — Engineer 15

"If you're going to come up with a fix, you're going to need a lot more knowledge than what the tool tells you." — Engineer 14

"It depends on the type of error. [...] Explaining the error in just one line of text can sometimes be more confusing than helping." — Engineer 15

"With this example, if you didn't mention the path between them, you'd look at those balises, and wonder 'how do I get between them?' But you might think of that anyway." — Engineer 14

"If you're just doing one thing, then it's fine; but the problems can be related to each other." — Engineer 4

The designs for the subsequent tasks were largely similar to the common design from task 1, however in task 3 (Figure 6.6), group 1 pointed out that the exercise sheet did not contain enough information for a user to understand the error: A "VPCA" distance is measured from a signal or marker board, opposite to the direction of travel; however, in this case there were no infrastructure elements that could be used as reference.

“A node doesn’t tell me which direction the VPCA applies to. If I had a marker board, I would know. These nodes are just names in the [REDACTED] model. They don’t actually mean anything to a signalling engineer, who needs route-setting points.” — Engineer 6

“In order to evaluate that error, we need more information than is shown at the moment. We need to know what that VPCA start is for.” — Engineer 10

“The problem is that the VPCA start isn’t a physical object, but an offset from something (e.g. signal), so you probably need to show where that object is, so that the user can understand why the balise should be there.” — Engineer 0

6.1.4 Consensus on Counterexample Representation

After the focus group session, we transcribed the video recording, then analysed the transcript. We summarise the consensus we reached with the rail engineers below.

Visualisation Design We use the following principles to design the visualisations of counterexamples:

1. Use a dark orange colour to show the attributes of objects directly involved in the violation, e.g. the distance between a balise and an existing piece of infrastructure.
2. Use blue to highlight objects and attributes related to the error. In case the error involves both “new” (balises) and “old” (e.g. points) equipment, we do not make the distinction as to which is the cause of the problem, or which is easier to change — the decision on how to fix the error is for the engineer to make.
3. Use a semi-transparent circle for node-like objects (e.g. balises, AWS ramps, points), and a semi-transparent rectangle for portions of a track (e.g. to show a problematic distance, or an area related to the error)

In addition to the graphical representation, we also provide a textual description for each type of violation. We formulate this text in the following style:

1. The subject of the sentence is the violating attribute(s) (e.g. the distance between two objects).
2. This is followed by the names of the objects involved with their locations.
3. At the end of the sentence, we show the required and actual values of attributes, if any.

This description is shown along with a link to the original text of the rule.

Interaction details

- Our scope in this project is to check an entire scheme plan against one design rule at a time. In the future, we would like to extend the [REDACTED] further to check fragments (areas of interest), and against multiple rules at the same time.

- If we implement multi-rule checking, it would be essential to have a mechanism for bundling rules based on a classification scheme like the one we introduced (subsubsection 6.1.2.1).
 - The user can then select which bundles to check for a specific stage in the design process. For example, in a certain stage, they may wish to check all the rules concerning distances between objects.
 - We need to consider how to represent two violations of a similar nature (e.g. both involving distances between objects) but of different rules, so that they do not get mistaken for violations of the same rule.
- Contextual information (textual and graphical) is important to enable the mapping between the digital presentation and the real world. Details such as layout orientation, while not affecting the scheme plan data, should match real-world scheme plans to make the visualisation intuitive.

Interaction Sequence Based on the interaction design from group 3 and the plenary discussion, we think the user experience of the visualisation should follow this sequence, when multi-rule checking and visualisation is implemented:

1. A list of short descriptions is shown in a pop-up window (one per counterexample), which can be sorted by rule or by object name.
2. When the user clicks on a counterexample, the relevant area is highlighted.
3. Within the area, each object involved is labelled with the number of violations found in the check.
4. The user clicks on an object to display a list of violations
5. When a specific violation is chosen, a detailed visualisation is shown for that object, and all other highlights are removed. This also brings up a detailed description of the violation and a reference link to the design rule.
6. The user can mark a violation as resolved, which hides it from the visualisation.

6.1.5 Evaluation of the Focus Group Study

The focus group enabled us to reach a consensus with experts in the railway domain on visualisation designs, which is what we hoped for; however, we were surprised that the participants considered the design of the workflow to be more important than the design of the visualisations. It was valuable to think of our work in the broader context of the [REDACTED] / [REDACTED] toolkit, and how our customers (engineers in delivery teams) want the toolkit to fit into their day-to-day work. Many issues discussed involved not just the [REDACTED] but how it works in relation to other tools, as well as the information used as reference when developing these tools (such as the wording of design rules). In the long term, we think it will be helpful to regularly involve stakeholders — customers and developers of other [REDACTED] projects — to

gather ideas on the design of new features in the [REDACTED] such as displaying violations of multiple rules at once.

The discussion during the focus group uncovered some issues with the material we provided. Firstly, in the examples we produced, we only showed elements directly involved in the errors, and not ones that provide important contextual information (such as a signal or marker board for task 3); secondly, we did not use standard scheme plan symbols, which meant the engineers had to map our custom symbols to the ones they are used to. However, we would like to remark that this was largely due to the fact that common knowledge of the railway industry (e.g. symbols and jargons) are not easily accessible to people who do not work directly in the railway industry, despite not having any commercial sensitivity.

Our moderation of the focus group session could also be improved. A large proportion of the time in the plenary discussion was spent on design issues of other parts of the wider [REDACTED] "ecosystem", rather than being focused on the visualisation of counterexamples produced by the [REDACTED]. Although since we were on [REDACTED] premises as guests, we did not want to be overbearing with the moderation.

6.2 Our Proposed Designs for Representing Single Errors

With the feedback from our focus group in mind, we now present our proposed designs for the rendering and description of the counterexamples in Table 6.1 below. Note that we only show designs for single violations in each category – the presentation or 'bundling' of multiple violations is outside the scope of this section. These designs were the results of an agreement with the [REDACTED] project supervisors.

Table 6.1: Proposed Visualisations of Counterexamples

Rule	Variant	Editor Display	Counterexample	Textual Description

Table 6.1: Proposed Visualisations of Counterexamples

Rule	Variant	Editor Display	Counterexample	Textual Description

Table 6.1: Proposed Visualisations of Counterexamples

Rule	Variant	Editor Display	Counterexample	Textual Description

Table 6.1: Proposed Visualisations of Counterexamples

Rule	Variant	Editor Display	Counterexample	Textual Description

Chapter 7

Software Architecture and Design

Contents

7.1	Overall Process	56
7.2	Design Considerations	60

Several programs and libraries are involved in the process of checking scheme plan data against design rules. In this chapter we will provide an overview of the interaction between the software components, and our modifications to the software to enable counterexample visualisation.

Preliminary Concepts

In this chapter we will reference some concepts in software development, which we explain below:

Backward Compatibility We say a change to a piece of software is *backward compatible* if data or programs designed to work with an older version of the software will have the same behaviour on the newer version.

Loose and Tight Coupling In the context of this chapter, we say two software components are *loosely coupled* if the correct behaviour of one component depends only on the public API of the other (e.g. the signature of a function). Conversely, *tightly coupled* components rely on knowledge of implementation details to work correctly.

Sequence and Class Diagrams These are diagrams drawn to our understanding of the Unified Modelling Language [24]. Sequence diagrams describe communications between software components, and class diagrams describe data structures and relations between data types.

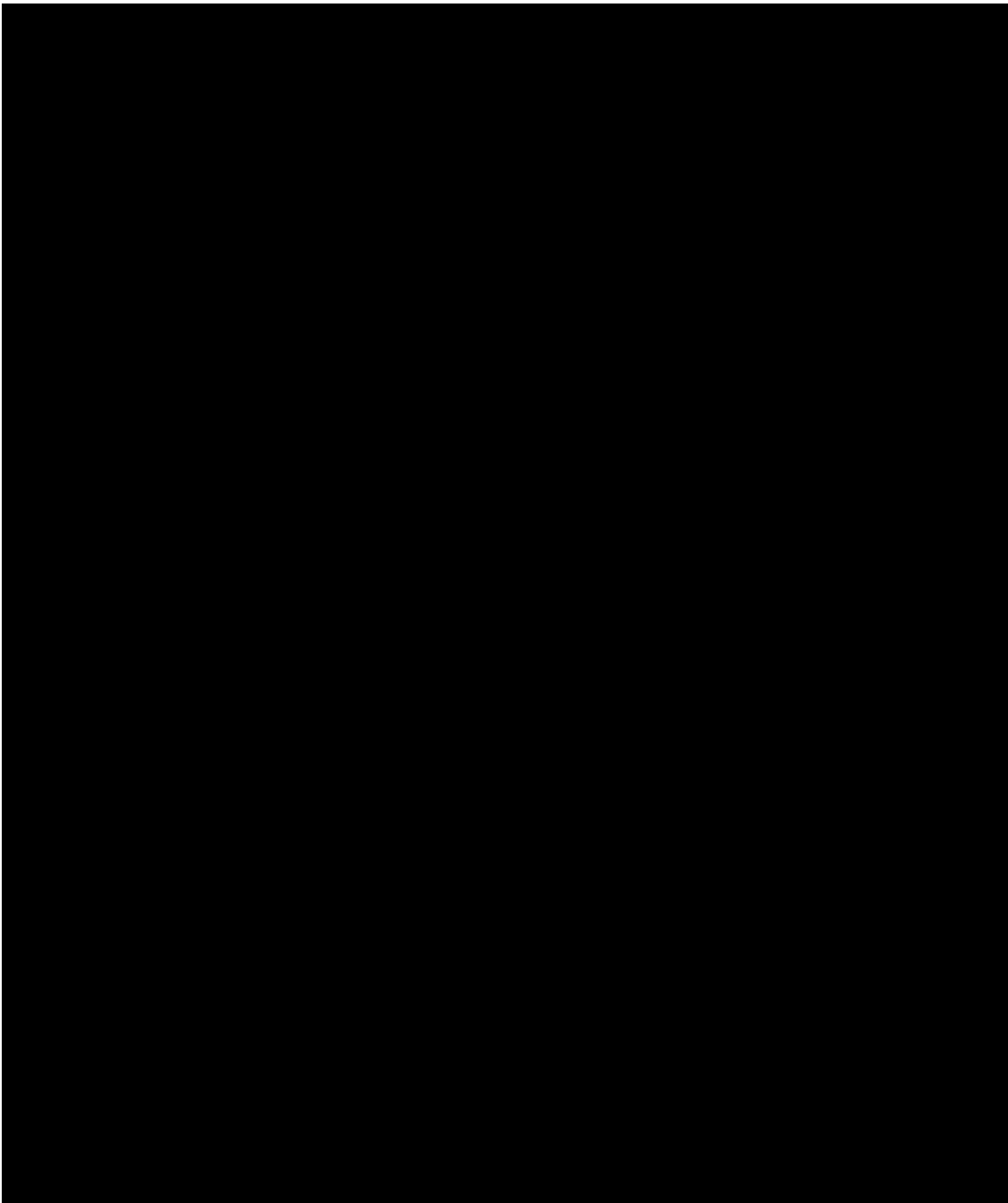


Figure 7.1: Sequence of messages between the tools when the [redacted] returns a *Pass* or *Unknown* result. We use the UML **alt** operator to denote possible outcomes of the same action. Note that the `getreport` command will only be issued (and retrieve one of the possible verification results) if the job has completed.

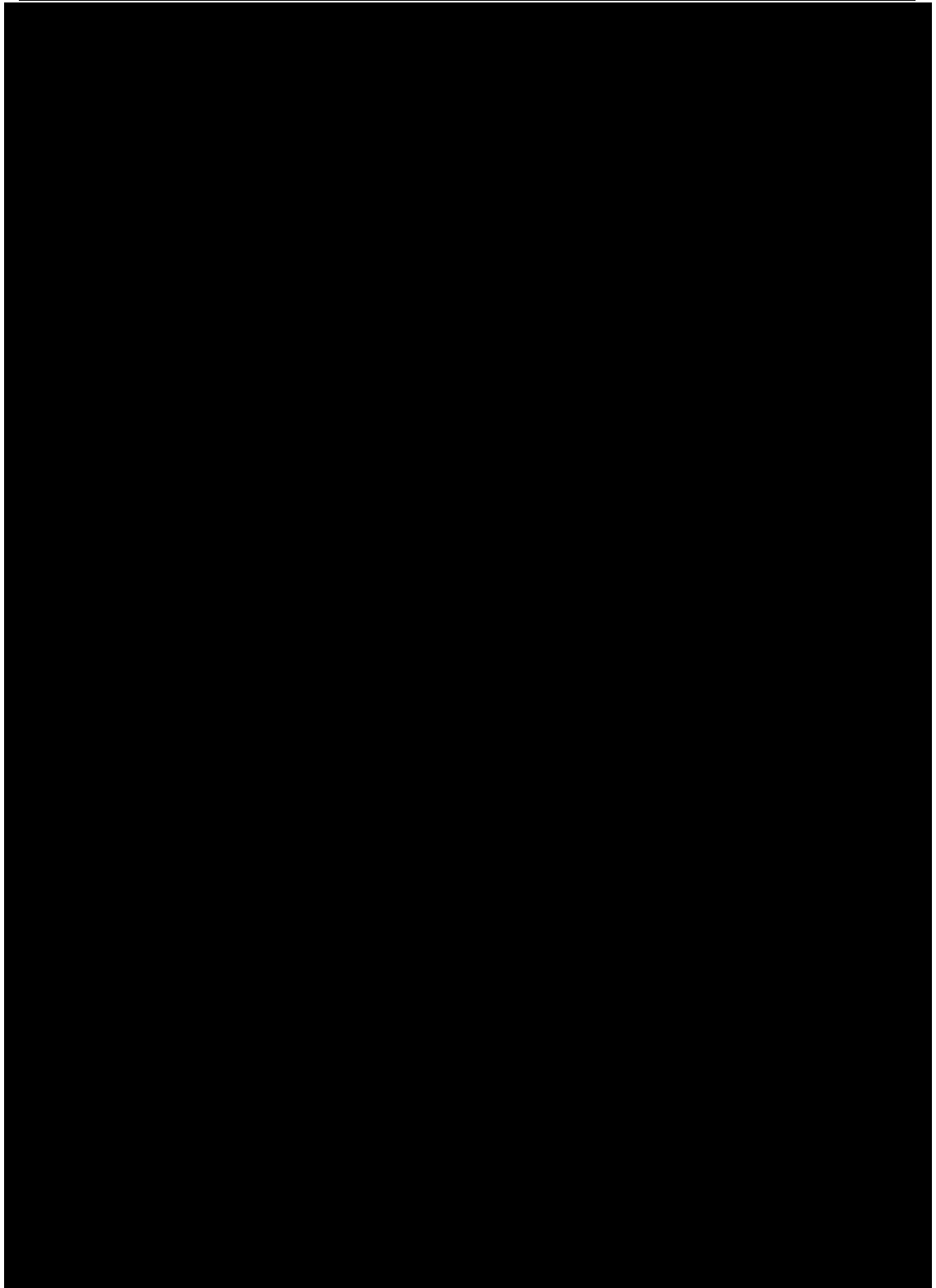


Figure 7.2: Sequence of messages between the tools when the [redacted] returns a *Fail* result. Our modifications are shown in red.

7.1 Overall Process

Recall from Section 4.3 that the [redacted] can return one of three outcomes when verifying a scheme plan against a single design rule:

- *Pass* means the [redacted] was able to prove that the scheme plan complies with the design rule.
- *Fail* means the [redacted] found a violation of the design rule.
- *Unknown* means the [redacted] was unable to determine if the scheme plan complies with the design rule.

We are interested in improving the *Fail* case to visualise counterexamples.

Figures 7.1 and 7.2 show the interactions between the end user, [redacted] and the [redacted] in the three aforementioned cases after our modifications. After the user opens a set of scheme plan data ([redacted] files, see Section 5.1.1) in [redacted] and switches to the “[redacted] [redacted] [redacted]” tab, the following set of events occur:

1. Start of verification:

- a) The user clicks the “Verify” button in [redacted]
- b) [redacted] sends a `verify` command to the [redacted] client, along with some command line arguments:
 - [redacted]: The path to the [redacted] files.
 - `designrulepath`: The directory containing XML files describing design rules.
 - `designrules`: The IDs of the design rules to be checked.
 - `outputpath`: The location to be used for the [redacted]’s report.
- c) The [redacted] client sends this data verbatim to the server as a REST API call.
- d) The [redacted] server sends a REST API response indicating that the verification job has started. This response contains metadata about the job, notably its *job ID* and *report ID*, the latter of which will later be used to retrieve the verification result.
- e) The [redacted] client serializes the response from the server into an XML file at the location specified by the `outputpath` option.
- f) [redacted] retrieves the report ID from the XML file, and indicates to the user that the verification has started.

2. **Checking verification progress:** While the verification job is running, the user can click the “Refresh” button in [redacted] to check its status. In this case the [redacted] client receives a `getjob` command¹ with the *job ID* from the start of verification, and sends a query to the server with this ID. The job status it receives from the server is then written to an XML file in the location previously specified. If the job is still in progress, [redacted]

¹A “job” refers to a task the [redacted] has been given, such as verifying a set of [redacted] data against a design rule.

indicates this to the user; if the job has finished, we proceed to retrieve the verification result.

3. End of verification:

- a) [redacted] issues a `getreport` command to the [redacted] client, with the *report ID* as a parameter.
- b) The [redacted] client sends the report ID to the server to retrieve the report for the verification job, containing the verification result, and in case of a *Fail* result, information that can be used to describe and visualise counterexamples.
- c) The [redacted] client serializes the report it received into an XML file.
- d) [redacted] indicates to the user the result of the verification.
- e) If the result is *Fail*, [redacted] shows a textual description of the violation, and a “Show Counterexample” button is made available to the user.

4. Retrieving and visualising counterexamples:

- a) The user clicks the “Show Counterexample” button in [redacted].
- b) [redacted] sends the `get` [redacted] command to the [redacted] client with the report ID and the path of the [redacted] data directory given as command-line arguments.
- c) The [redacted] client retrieves a report from the server in the usual fashion.
- d) Using the information in the report, the [redacted] client writes additional [redacted] — containing counterexample information — to the specified directory.
- e) [redacted] reloads the scheme plan data from disk, and displays the counterexample information as a graphical overlay in the “Track Layout” tab.

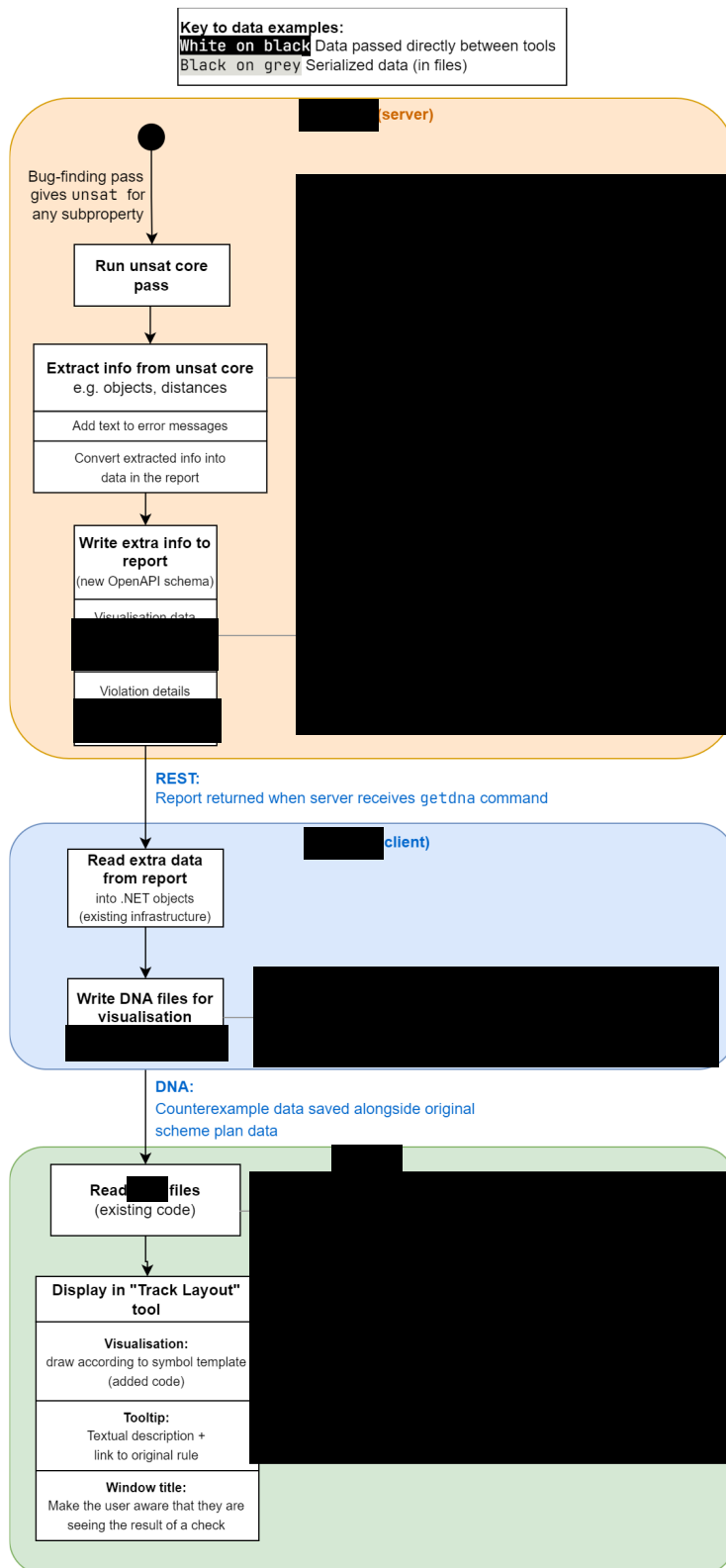
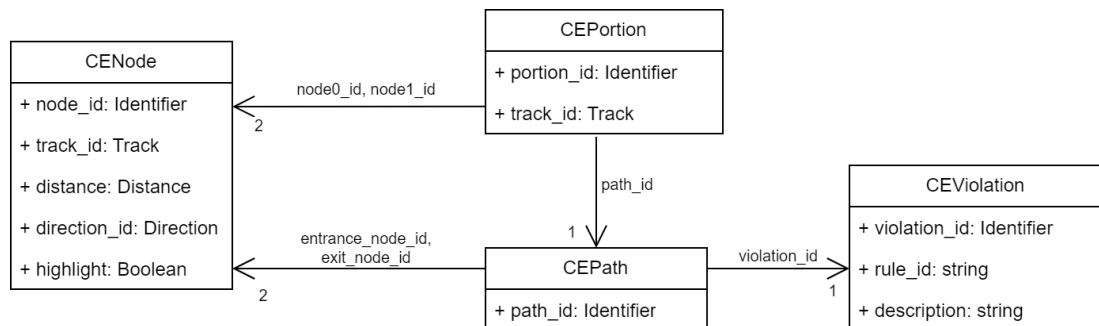
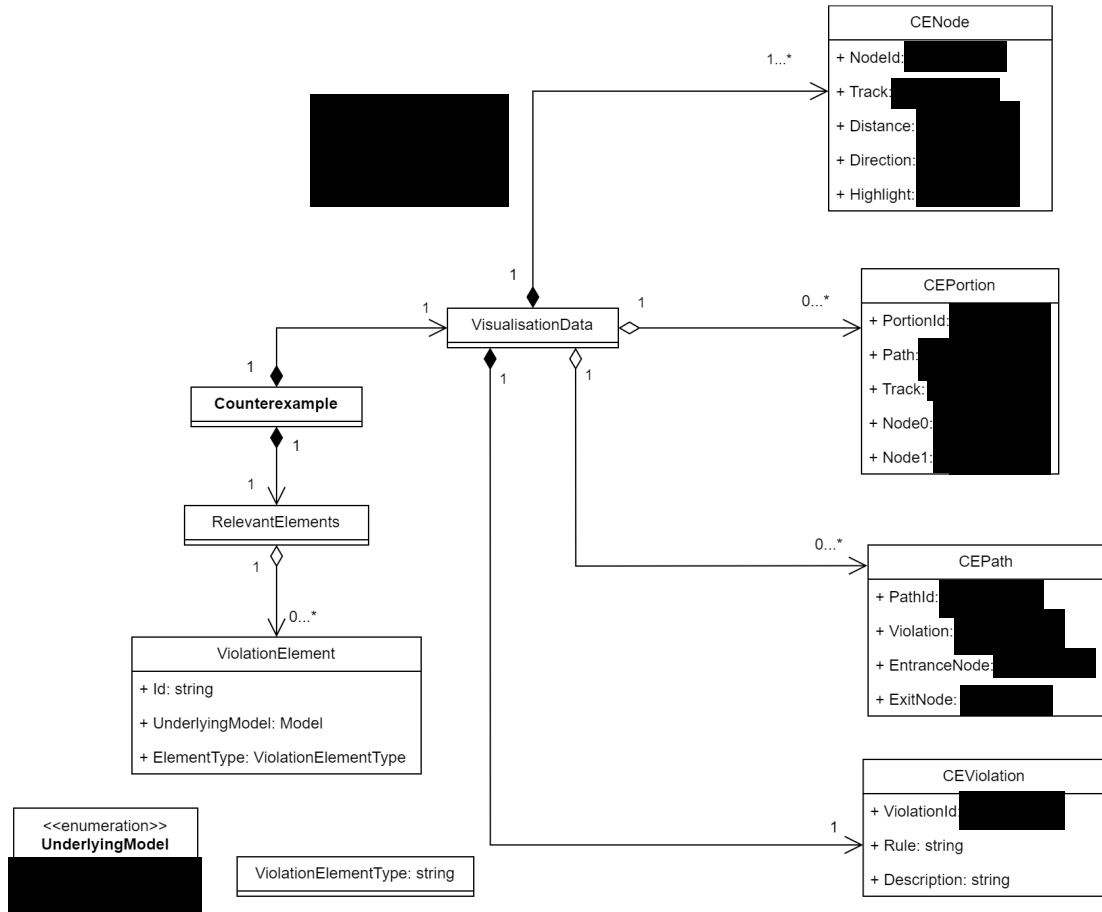


Figure 7.3: The overall transformation process for counterexample data.

As shown in Figure 7.2, our work mostly concerns generating, retrieving and displaying information about counterexamples. This requires transforming data across several data models. Figure 7.3 presents a more detailed view of the data transformation in the last message sequence of Figure 7.2. We will discuss each of the steps in detail in the following chapters.



7.2 Design Considerations

In this section, we describe engineering principles we adopt in software development, and how we follow them in practice.

7.2.1 Introducing Data Types

Visualising counterexamples requires generating and transforming extra data across the [REDACTED] server, the client, and [REDACTED]. The extra data consists of two parts:

- *Visualisation Data* which will be graphically displayed in the “Track Layout” tab of [REDACTED]
- *Relevant Elements* containing identifiers of TDM or NEM elements that were involved in a violation. This information will be displayed elsewhere in [REDACTED], such as in error messages.

As our project is a proof-of-concept for counterexample visualisation, we have decided to focus on implementing the transformations of the *Visualisation Data* in our limited timescale. Every set of visualisation data must contain exactly one *CEViolation* — which contains meta-data about the violation — and at least one *CENode*, the “base unit” that represents a position on the scheme plan. A *CENode* can be highlighted (shown over the existing plan) if it represents an object involved in the violation, or hidden if it is used to define one end of a *CEPortion*, which represents a segment of track. *CEPortions* can in turn be grouped together to form *CEPaths* that may extend across multiple *CENodes*.

Because this toolchain is a distributed system in which we cannot guarantee that all components are at the latest version, we need to take particular care when making changes to the data interfaces between them:

- The report generated by the [REDACTED]’s server and delivered to the client over its REST API (Figure 7.4).
- The [REDACTED] generated by the [REDACTED]’s client for visualisation in [REDACTED] (Figure 7.5).

Our changes to these interactions between tools are *additive* to ensure backward compatibility of both the data and the tools. This means:

- We never modify existing data models, only add extra data types. If an older version of a tool is given data with elements we added, it will behave as if such elements were not present.
- Our modifications to the logic of how the data interfaces only concern the data types we add, so the existing behaviour of the tools are preserved. This way, data that was created prior to our changes will continue to be handled correctly by the newer versions of the tools.

Whenever possible, we implement our modifications to the data models in a way that is compatible with the existing data generation / parsing infrastructure with no or minimal

changes. For example, the additional data given to [redacted] for visualisation is written into [redacted] files using existing functions in [redacted], and the files are parsed by [redacted] in the same way as it would for any set of [redacted] data. By concentrating our modifications on the data models (schemas) rather than the logic that handles the data, we reduce the chances of introducing bugs in the tools or breaking compatibility.

The [redacted] client interfaces with both the REST report from the [redacted] server and the [redacted] files to be passed onto [redacted], which makes it especially susceptible to mistakes in implementing data transformation. As a result, we minimise its complexity by making the data on both sides directly correspond with each other — the data we put into the report is exactly the data to be written into [redacted] files, and the client only needs to parse the data it receives into arguments to existing functions in [redacted].

The consequence of this design is that the bulk of the data transformation resides in the [redacted] server, which has access to the original TDM data, the NEM data, the representation of NEM data in SMTLib, as well as the mapping of identifiers between these data models. This gives us the flexibility to implement specialised logic for different types of counterexamples in the future: so long as the data model used in the report (and [redacted] data) does not change, we only need to make modifications in the [redacted] server to display a different combination of visual elements in [redacted].

7.2.2 Breaking Large Operations Into Modular Functions

The data transformation process contains several operations where complex data structures need to be constructed. Doing an operation like this all in one long function can lead to the logic being difficult to follow due to the large amount of data in the function's state. This also makes debugging harder in case the function does not behave as intended. To address this risk, we adopt a style of breaking down these large operations into one static local function for each step [50]. This brings us the following benefits:

- Splitting the operation into smaller functions reduces the levels of indentation and makes the code easier to read.
- By placing the functions within the function that carries out the overall operation, we make them inaccessible to anything outside the enclosing function's scope, and clearly indicate that the code within the local function is only used in the enclosing function.
- By declaring the local functions `static`, we isolate them from the state of the enclosing function. Anything the local function needs to use from the enclosing state is passed as a parameter (i.e. it is free of referential dependencies), and anything it produces is returned (i.e. it is free of side effects). This also improves readability because the dependencies and effects of each step can be directly inferred from the signature of the local function.

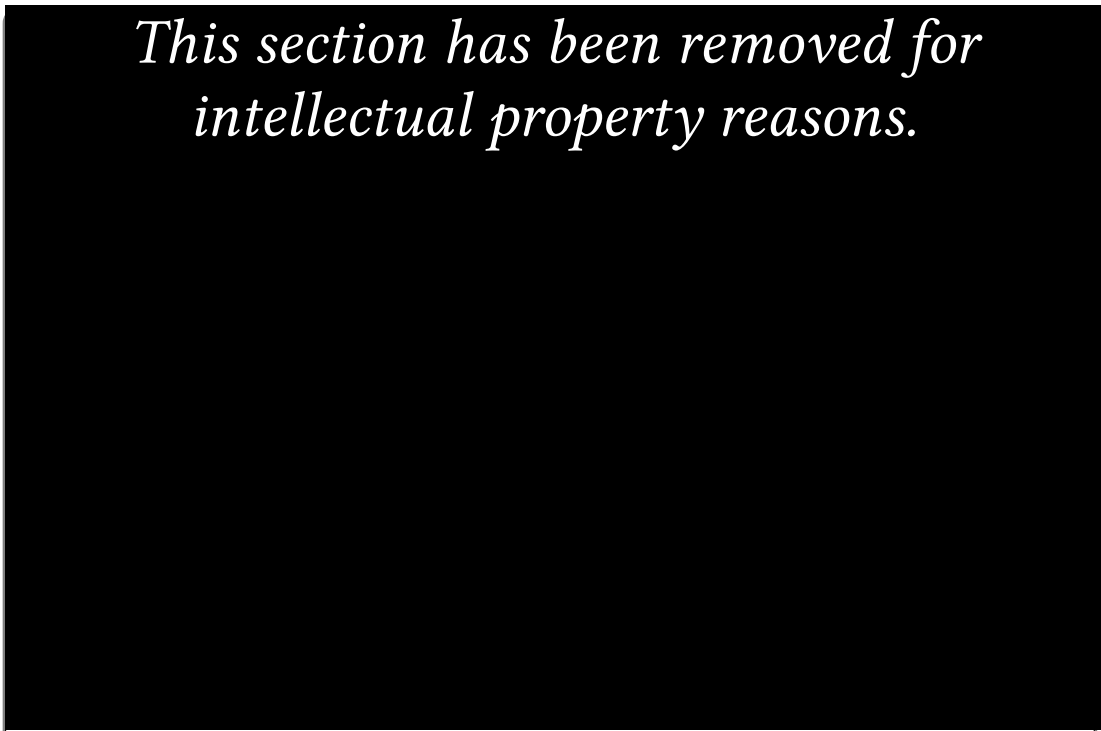
However, this approach also has its drawbacks. In particular, passing data as parameters and return values can introduce a performance cost compared to directly manipulating local variables in the enclosing function. We consider this cost to be relatively negligible due to

the lightweight nature of reference types in C# [44], — which are the majority of data we pass around — and justified by the benefits outlined above.

7.2.3 Controlling Mutability of State

Generally speaking, passing data between program components in a read-only form helps make the logic of the program easier to follow, since we have more certainty in the contents of arguments passed to functions and values they return [23]. In C#, arguments are passed to functions *by value* by default [51]. For a function parameter of value type, this means any modifications to that parameter is contained within the function, because the function receives a *copy* of the argument provided by its caller. However, for a parameter of reference type, whilst the *value* (memory address) of the argument is copied into the function, the underlying *data members* of the argument can still be modified using its public fields or mutator methods.

This can be problematic for collection types such as `List` [49], which have methods that modify the collection's content: even if a function is declared `static` in the fashion we previously described, it can still have side effects on its surrounding scope by calling mutator methods such as `Add()` and `Remove()` on its collection parameters. To minimize unintended side effects, unless a function specifically needs to modify the contents of a collection parameter, we pass and return collection data using the `IReadOnlyCollection` interface [48], which remove the mutator methods from the API of the underlying collection.



*This section has been removed for
intellectual property reasons.*

*This section has been removed for
intellectual property reasons.*

7.2.4 Packaging

Previously [redacted] was a library used only by [redacted], so it was included in the [redacted] codebase as a Git submodule [11]. This means the [redacted] repository has a `.gitmodules` file that contains the hash (ID) of the revision of the [redacted] repository to be used when building [redacted]. Continuing this practice in our development of the [redacted] would have two

disadvantages:

- Git revision hashes are difficult to keep track of, because their ordering cannot be directly determined without referencing the history of a repository.
- Git submodules tightly couple components: Because [redacted] depends on [redacted] on a source level, if we want to incorporate any changes made to [redacted] into [redacted] we would need to both update the submodule commit hash in the [redacted] repository, and recompile [redacted].

To mitigate these effects, we package [redacted] as a NuGet package [47] and include it as a binary library dependency of both [redacted] and the [redacted] where the version of the library is explicitly specified. As long as we don't make backward-incompatible changes to the library's API, no code changes or recompilations are needed for the tools to take advantage of improvements in the library [55].

Chapter 8

Modifications to the [REDACTED]

Contents

8.1	Adding the Counterexample Pass to the Verification Algorithm	67
8.2	Extracting Information from the Unsat Core	70
8.3	Constructing Counterexample Data	73
8.4	Saving Counterexample Data	79

The [REDACTED] comprises server and client components, as described in Section 5.2.1. Our work involves modifications to both components, broadly as follows:

Server Changes

- Addition of a *counterexample pass* to the verification algorithm.
- Extracting information from the unsat core returned by Z3.
- Using the extracted information to construct counterexample data to be included in the report.

Client Changes

- Implementation of a `get[REDACTED]` that writes counterexample data from the report into `[REDACTED]` files.

The rest of this chapter will explore each of these points in detail.

A Note on Namespaces The [REDACTED]'s code base is organised into modules that each live in a separate namespace [53]. We make references to these namespaces in code samples throughout this chapter. Since all namespaces for the [REDACTED] start with [REDACTED], we omit this common prefix when referring to classes, methods and functions — for example, where the fully qualified name for a class is [REDACTED].`RESTServer.Verification.VerificationJob`, we will refer to it as `RESTServer.Verification.VerificationJob`.

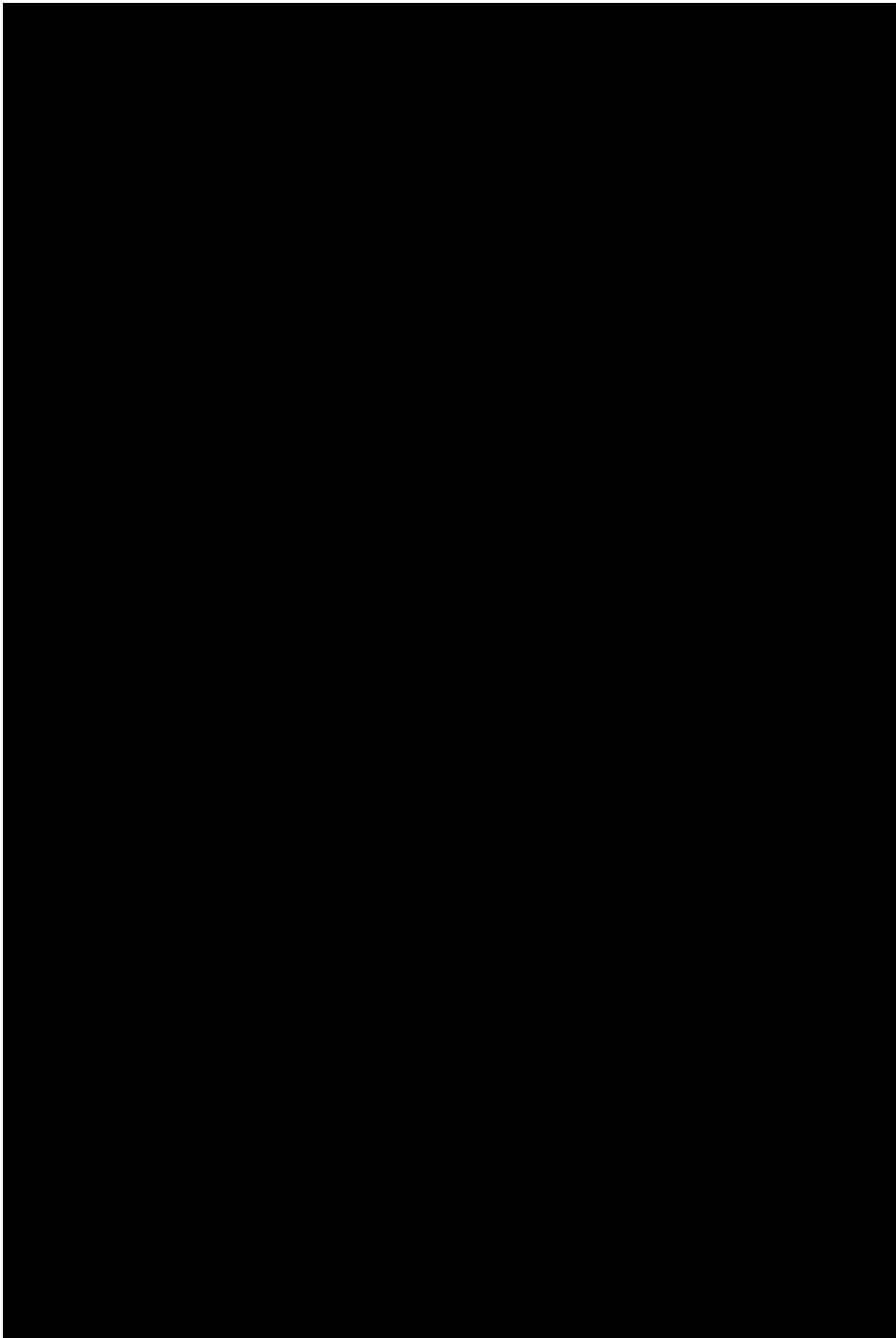


Figure 8.1: An overview of the server-side operations involved in producing counterexample data.

8.1 Adding the Counterexample Pass to the Verification Algorithm

The existing algorithm in the `verify` function for verifying data, as described in Section 4.3, does not obtain an unsat core from Z3 in case the first pass gives an unsat result. The information in the unsat core is essential for visualising counterexamples, so as a first step we add an extra pass to the algorithm.

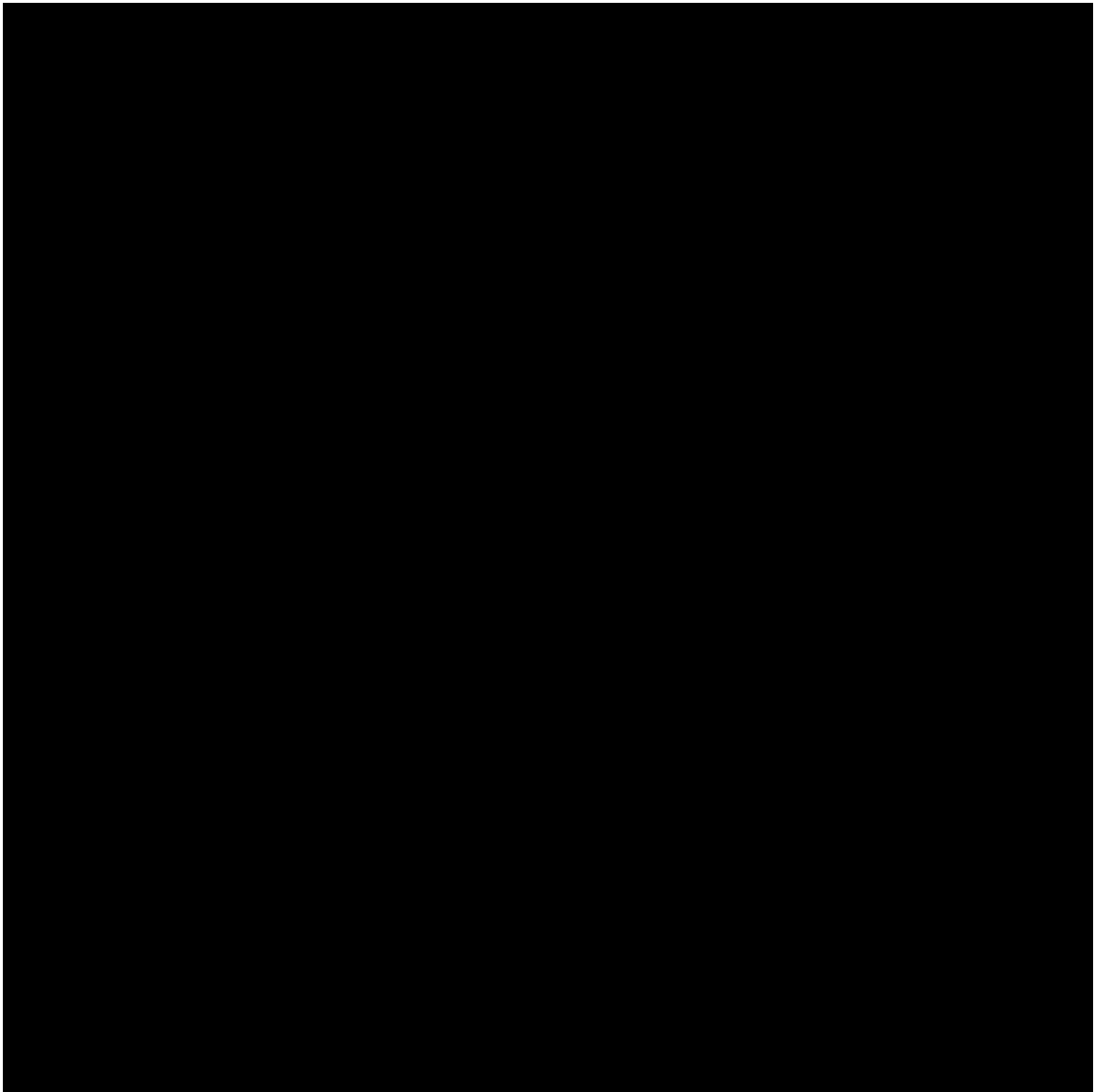


Figure 8.2: The `verify` function's verification algorithm after our modifications.

8.1.1 The Existing Procedure

At the beginning, the `Verifier`'s verification algorithm is given an SMTLib2 script with a *prelude* containing common definitions used across scheme plans (such as functions for calculating distance and reachability), followed by *subproperties* which are formulae that represent design rules instantiated with applicable elements of the scheme plan. The detailed procedure of the `Verifier`'s algorithm is described in [3]. Notably, it is discussed that if the algorithm says a scheme plan conforms with a design rule, this result is correct.

The algorithm consists of the *bug-finding pass* and the *validating pass*. Here, we are taking a view independent of the algorithm, that if in the *bug-finding pass* the SMT solver says a property does not hold, then a counterexample has been found. From such a result, we will describe in the following how a counterexample can be transformed to a visualisation.

8.1.2 Implementing the Counterexample Pass

The revised algorithm is illustrated in Figure 8.2 and the overall programmatic steps involved in our implementation are shown in Figure 8.1. When the bug-finding pass finds a violation, we run our additional *counterexample pass*, which repeats the procedure of the bug-finding pass but with only the subproperties that previously returned `unsat` and with the `:produce-unsat-cores` option turned on.

The existing implementation uses the visitor pattern [27] to write the SMTLib2 script to a temporary file, and we conform to this pattern in our extension to ensure backward compatibility. Our extension is implemented in three components. First, we define a `SMTLib.Script.GetUnsatCore` class that contains constants for adding a `get-unsat-core` command. The marker is to be printed above the `unsat` core in the output, so that it becomes easier to extract the contents of the `unsat` core later:

```
public static readonly string GETUNSATCORE_COMMAND_STRING = "get-unsat-core
    ↪ ";
public static readonly string UNSAT_CORE_MARKER = "UNSAT CORE:";
public static readonly string UNSAT_CORE_MARKER_COMMAND = $"echo \"{
    ↪ UNSAT_CORE_MARKER}\"";
```

Next, we implement an overload of the `SMTLib.ScriptVisitor.ToStringVisitor.Visit()` method to add the commands to the script in order:

```
1 public IExpression Visit(GetUnsatCore command)
2 {
3     builder.Append(L_BRACKET);
4     builder.Append(GetUnsatCore.UNSAT_CORE_MARKER_COMMAND);
5     builder.Append(R_BRACKET);
6     builder.Append(NEW_LINE);
7
8     builder.Append(L_BRACKET);
9     builder.Append(GetUnsatCore.GETUNSATCORE_COMMAND_STRING);
10    builder.Append(R_BRACKET);
```

```

11     builder.Append(NEW_LINE);
12     return command;
13 }

```

Finally, the `SMTLibTranslator.Translator.SMTScriptWithPrelude.ToString()` method contains the top-level logic for writing commands to a script, and here we add a case for the counterexample pass:

```

1  builder.Append("(push)");
2
3  switch (Variant)
4  {
5      case CheckVariant.Validating:
6          // ... (commands for the validating pass)
7          break;
8      case CheckVariant.UnsatCore:
9          // ... (same commands as the bug-finding pass)
10         GetUnsatCore unsatCoreCommand = new();
11         unsatCoreCommand.Accept(new ToStringVisitor(builder));
12         break;
13     case CheckVariant.BugFinding:
14         // ... (commands for the bug-finding pass)
15         break;
16     default:
17         throw new ArgumentOutOfRangeException("Invalid check variant");
18 }
19
20 builder.Append("(pop)");

```

This results in the following two commands being added to the script for the counterexample pass, after the usual check-sat command for each sub-property:

```

(echo "UNSAT CORE:")
(get-unsat-core)

```

The counterexample pass of the verification algorithm carries out three steps:

1. Go through the previously parsed output from Z3 and put any subproperties that gave unsat into a new SMTLib2 script, with additional commands to obtain the unsat core as described above.
2. Run the new script and parse the output of Z3 with the `RunScript()` method. Each sub-property produces a `SolverResult` consisting of the output of the check-sat command (which in this case will always be unsat), and a set of counterexample data extracted from the unsat core. The `RunScript()` method returns a list of `SolverResults`, in the same order as the subproperties in the SMTLib2 script. We use the `Zip()` method to create a mapping between each sub-property and its `SolverResult` [57].

3. Finally, we copy the counterexample data into the verification results (which can be *Pass* / *Fail* / *Unknown* for each sub-property, but in this case will always be *Fail*), to be later used when creating the report.

The following code listing shows the implementation of this additional pass:

```

1 script.Variant = SMTScript.CheckVariant.UnsatCore;
2 script.RemoveAllSubProps();
3
4 // Step 1: Put previously-unsat subproperties in the script
5 foreach (var (subprop, result) in solverResults)
6 {
7     if (result.Response is ISolver.CheckSatResponse.Unsat)
8     {
9         checkResults[subprop] = new Result { SubpropertyResult =
10             ↪ DesignRuleCheck.ResultEnum.Fail };
11         script.AddSubProp(subprop);
12     }
13 }
14 // Step 2: RunScript() parses the output of Z3 and populates the CeData in
15     ↪ each SolverResult
16 Dictionary<Subproperty, ISolver.SolverResult> unsatPassResults =
17     script.Subproperties.Zip(
18         solver.RunScript(script),
19         (subprop, result) => new { subprop, result }
20     ).ToDictionary(x => x.subprop, x => x.result);
21 // Step 3: Put the counterexample data in the verification result for each
22     ↪ subproperty
23 foreach (var (subprop, result) in unsatPassResults)
24 {
25     if (result.CeData is null)
26     {
27         throw new Exception($"Subproperty was in unsat core pass, but
28             ↪ counterexample data was null: {subprop}");
29     }
30     checkResults[subprop].CeData = result.CeData;
31 }

```

8.2 Extracting Information from the Unsat Core

Once we run the counterexample pass, we need to extract information from the unsat core. We implement this functionality in the method with direct access to the SMTLib2 script

and the output from `Z3 — SMTLib.Solver.Z3Solver.RunScript()` — as self-contained local functions in the style we described in Section 7.2.2.

Each category of design rule requires a different representation of its counterexamples, so the first step is to sort design rules into categories (as we described in subsection 6.1.2.1) based on their IDs:

```

1 static ISolver.RuleCategory GetRuleCategory(string designRuleId)
2 {
3     var match = RulePattern.Match(designRuleId);
4     string ruleName = match.Groups["name"].Value;
5     string variant = match.Groups["variant"].Value;
6
7     return (ruleName, variant) switch
8     {
9         ("BG-01", _) => ISolver.RuleCategory.Distance,
10        ("BG-02", "1")
11            or ("BG-02", "2") => ISolver.RuleCategory.Distance,
12        ("BG-02", "3") => ISolver.RuleCategory.ProhibitedPlacement,
13        ("BG-03", "1")
14            or ("BG-03", "2") => ISolver.RuleCategory.Distance,
15        ("BG-03", "4") => ISolver.RuleCategory.ProhibitedPlacement,
16        ("BG-05", "") => ISolver.RuleCategory.Separation,
17        _ => throw new NotImplementedException("Design rule is not supported
18            ↪ for counterexample visualisation")
19    };
20 }

```

Next, we parse the unsat core for information based on the rule category. For our proof of concept, we handle the “distance” category and extract NEM edge names using a regular expression:

```

1 private static readonly Regex EdgePattern = new(@"def_edge_(^[^s()]+)");
2
3 // Populate the `CeData` field of a result based on the unsat core in the
4 // ↪ output.
5 static void ParseUnsatCore(ISolver.SolverResult result, string outputLine,
6     ↪ DesignRule? designRule)
7 {
8     // ... (input validation)
9
10    var ruleCategory = GetRuleCategory(designRule.Id);
11
12    if (ruleCategory is ISolver.RuleCategory.Distance)
13    {
14        var edgeMatches = EdgePattern.Matches(outputLine);
15    }
16 }

```

```

13
14     if (edgeMatches.Count <= 0) return;
15     foreach (Match match in edgeMatches.Cast<Match>())
16     {
17         string edgeName = match.Groups[1].Value;
18         ceData.EdgeNames.Add(edgeName);
19     }
20 }
21 }

```

Finally, we incorporate the parsing of unsat cores into the overall logic of parsing the output of Z3, making use of the unsat core marker we added:

```

1 static IReadOnlyCollection<ISolver.SolverResult> ParseZ3Output(StreamReader
    ↪ outputStream, DesignRule? designRule)
2 {
3     List<ISolver.SolverResult> results = new();
4     bool currentLineIsUnsatCore = false;
5     while (!outputStream.EndOfStream)
6     {
7         string outputLine = outputStream.ReadLine();
8         if (currentLineIsUnsatCore)
9         {
10             Log.Information($"Unsat core: {outputLine}", LogClient.XXXXXXXXXX);
11             ParseUnsatCore(results.Last(), outputLine, designRule);
12
13             currentLineIsUnsatCore = false;
14         }
15         else if (outputLine.Equals(GetUnsatCore.UNSAT_CORE_MARKER))
16         {
17             currentLineIsUnsatCore = true;
18         }
19         else if (!string.IsNullOrEmpty(outputLine))
20         {
21             results.Add(outputLine switch
22             {
23                 "unsat" => new(ISolver.CheckSatResponse.Unsat, new ISolver.
                    ↪ CounterexampleData()),
24                 "sat" => new(ISolver.CheckSatResponse.Sat),
25                 "unknown" => new(ISolver.CheckSatResponse.Unknown),
26                 "error" => new(ISolver.CheckSatResponse.Error),
27                 _ => throw new NotImplementedException($"Unknown output from Z3:
                    ↪ {outputLine}")
28             });
29         }

```

```

30     }
31
32     return new ReadOnlyCollection<ISolver.SolverResult>(results);
33 }

```

8.3 Constructing Counterexample Data

When adding counterexample data to the report, we also start by defining the data types shown in Figure 7.4 in the OpenAPI schema.

8.3.1 Updating the OpenAPI Schema

Firstly, the `[redacted]` and `[redacted]` types reflect the value restrictions defined in `[redacted]` for identifiers and distances (i.e. valid values are subsets of strings and floating-point numbers, respectively):

```

[redacted]:
  type: string
  pattern: ^(?:[Nn]_)([a-zA-Z0-9_@.!\#()-/])*$
[redacted]:
  type: number
  format: float
  minimum: 0
  maximum: 1000000000
  multipleOf: 0.01

```

These are then used to define types used for visualisation, such as `CeNode` (corresponding to `CENode` in `[redacted]`):

```

CeNode:
  type: object
  properties:
    NodeId:
      $ref: '#/components/schemas/[redacted]'
    Track:
      $ref: '#/components/schemas/[redacted]'
    Distance:
      $ref: '#/components/schemas/[redacted]'
    Direction:
      $ref: '#/components/schemas/[redacted]'
    Highlight:
      type: boolean
  required:
    - NodeId

```

- Track
- Distance
- Direction
- Highlight

The collection of all visualisation elements, `VisualisationData` contains an array of each type of overlay element, plus a `Metadata` element that contains information such as the violation ID and the design rule ID:

```
VisualisationData:
  type: object
  properties:
    Nodes:
      type: array
      items:
        $ref: '#/components/schemas/CeNode'
      minItems: 1
      uniqueItems: true
    Portions:
      type: array
      items:
        $ref: '#/components/schemas/CePortion'
      minItems: 0
      uniqueItems: true
    Paths:
      type: array
      items:
        $ref: '#/components/schemas/CePath'
      minItems: 0
      uniqueItems: true
    Metadata:
      $ref: '#/components/schemas/CeViolation'
  required:
    - Nodes
    - Portions
    - Paths
    - Metadata
```

The `ViolationElement` type can be used to attach information about scheme plan objects involved in a violation to a counterexample. This information is not currently populated or passed on to [REDACTED], but can be used in the future to implement additional functionality — for example, allowing the user to navigate to the violating object, or to filter counterexamples by object type:

```
ViolationElement:
  type: object
```

```

properties:
  Id:
    type: string
  UnderlyingModel:
    type: string
    enum:
      - TDM
      - NEM
  ElementType:
    type: string
required:
  - Id
  - UnderlyingModel
  - ElementType

```

The top-level Counterexample type contains both the visualisation data and the collection of ViolationElements involved, and is placed alongside the result and error message for each sub-property under DesignRuleCheck:

```

DesignRuleCheck:
  type: object
  properties:
    instances:
      type: array
      items:
        $ref: '#/components/schemas/Instance'
        description: A concrete instance being used in the check
    result:
      type: string
      enum:
        - Pass
        - Fail
        - Unknown
        - Cancelled
        - Error
      description: The result of the particular design rule check.
    errorMessage:
      type: string
      description: Failure reason for the particular design rule check.
    Counterexample:
      type: object
      description: Data used to visualise and describe counterexamples.
      properties:
        VisualisationData:
          $ref: '#/components/schemas/VisualisationData'

```

```

RelevantElements:
  type: array
  items:
    $ref: '#/components/schemas/ViolationElement'

```

After we modify the OpenAPI schema, we use Swagger Codegen [89] to generate C# classes corresponding to our data types. These classes are shared by the server and the client to directly represent the contents of a report. For example, the generated C# class for CeNode looks like this:

```

1 public partial class CeNode : IEquatable<CeNode>
2 {
3     [Required]
4     [RegularExpression("^(^?![Nn_])([a-zA-Z0-9_@.!(#)-/])*$")]
5     [DataMember(Name="NodeId")]
6     public string NodeId { get; set; }
7
8     [Required]
9     [RegularExpression("^(^?![Nn_])([a-zA-Z0-9_@.!(#)-/])*$")]
10    [DataMember(Name="Track")]
11    public string Track { get; set; }
12
13    [Required]
14    [DataMember(Name="Distance")]
15    public decimal? Distance { get; set; }
16
17    [Required]
18    [RegularExpression("^(^?![Nn_])([a-zA-Z0-9_@.!(#)-/])*$")]
19    [DataMember(Name="Direction")]
20    public string Direction { get; set; }
21
22    [Required]
23    [DataMember(Name="Highlight")]
24    public bool? Highlight { get; set; }
25
26    // ... definitions of ToString(), Equals(), GetHashCode(), etc
27 }

```

8.3.2 Putting Data Into the Report

We define a `ProduceCounterexample()` function to convert the data previously extracted from an unsat core into the data types we use in the report. It uses the NEM data it gets passed to obtain values that should be put into the data elements under `Counterexample`. For the “distance” category of design rules we implement counterexamples for, this means looking up

edge names we extracted from the unsat core to obtain their start and end positions, where place CeNodes, and then constructing CePortions between them. Since each NEM object contains a reference to its TDM counterpart, we are able to use TDM identifiers as a basis for naming counterexample objects.

```

1  /// <summary>
2  /// Construct a counterexample to be included in the report.
3  /// This can then be serialised into [redacted] files for display in [redacted].
4  /// </summary>
5  /// <param name="ceData"> The counterexample data produced by the solver
   ➔ module. </param>
6  /// <param name="designRule"> The design rule being checked. </param>
7  /// <param name="errorMessage"> The error message for this violation. This
   ➔ should already have the element names filled in. </param>
8  /// <param name="model"> The NEM model containing all data for the scheme
   ➔ plan being checked. </param>
9  /// <returns></returns>
10 public static Counterexample ProduceCounterexample(
11     ISolver.CounterexampleData ceData,
12     DesignRule designRule,
13     string errorMessage,
14     NEMModel model)
15 {
16     var nodes = new List<CeNode>();
17     var paths = new List<CePath>();
18     var portions = ceData.EdgeNames
19         .Select(edgeName =>
20             model.EdgeList.Find(e => e.Id == edgeName))
21         .Select(edge => ConvertEdgeToPortion(edge, nodes, paths))
22         .ToList();
23
24     _violationCounter += 1;
25     var metadata = new CeViolation()
26     {
27         Rule = RuleNamePattern.Match(designRule.Id).Value,
28         Description = errorMessage,
29         ViolationId = _violationCounter.ToString()
30     };
31
32     return new Counterexample()
33     {
34         VisualisationData = new VisualisationData()
35         {
36             Metadata = metadata,

```

```

37     Nodes = nodes,
38     Portions = portions,
39     Paths = paths
40 },
41     RelevantElements = new List<ViolationElement>()
42 };
43
44 // Convert an NEM edge to a CE portion, and add its ends to the list of
45 // ↪ CE nodes.
46 // SIDE EFFECT: modifies parameters `nodes` and `paths`
47 static CePortion ConvertEdgeToPortion(Edge edge, ICollection<CeNode>
48 // ↪ nodes, ICollection<CePath> paths)
49 {
50     /* ... */
51 }

```

In addition, we define a function to add information to the existing error message based on the extracted counterexample data:

```

1  /// <summary>
2  /// Constructs an error message based on counterexample data
3  /// </summary>
4  /// <param name="ceData">Data parsed from an unsat core.</param>
5  /// <param name="model">Data model we use to look up elements referenced in
6  // ↪ the unsat core.</param>
7  /// <returns>The error message that corresponds to the data.</returns>
8  private static string ConstructErrorMessage(ISolver.CounterexampleData
9  // ↪ ceData, NEMModel model)
10 {
11     StringBuilder errorMessageBuilder = new StringBuilder();
12
13     // TODO: This method of adding up edge lengths can only be used for "
14     // ↪ distance" CEs
15     if (ceData.EdgeNames.Any())
16     {
17         decimal actualDistance = 0;
18         foreach (string edgeName in ceData.EdgeNames)
19         {
20             var edge = model.EdgeList.Find(e => e.Id == edgeName);
21             actualDistance += edge.Length;
22         }
23     }
24 }

```

```

21     errorMessageBuilder.Append($" (Actual distance was {actualDistance}m)
      ↪ ");
22 }
23
24 return errorMessageBuilder.ToString();
25 }

```

To put it all together, we add a step to the `RESTServer.Verification.VerificationJob.RunJob()` method — which constructs the report and has access to the verification results, the design rule, and the NEM data — with calls to the functions we defined. Note that here, `rulecheck` represents a `DesignRuleCheck` element in the report, and `checkresults[pro]` is the (internal) verification result of one sub-property.

```

1  if (checkresults[pro].CeData != null)
2  {
3      rulecheck.ErrorMessage += ConstructErrorMessage(checkresults[pro].CeData,
      ↪ currentModel);
4      rulecheck.Counterexample =
5          CounterexampleBuilder.ProduceCounterexample(
6              checkresults[pro].CeData, designrule, rulecheck.ErrorMessage,
7              currentModel);
8  }

```

8.4 Saving Counterexample Data

To retrieve and save the counterexample data from the server, the client needs to carry out three operations:

1. Get a report from the server.
2. Convert the counterexample objects in the server to [redacted].
3. Save the [redacted] to [redacted] files.

We implement this procedure in a new `get[redacted]` command, which takes two parameters using the *CommandLineParser* library [80]:

```

1  /// <summary>
2  /// Property containing the report ID.
3  /// </summary>
4  [Option('r', "report-id", HelpText = "The ID of the report that contains
      ↪ failed checks.")]
5  public string ReportId { get; set; }
6
7  /// <summary>
8  /// Path to the storage location of the [redacted] files.

```

8. Modifications to the [redacted]

```
9  /// </summary>
10 [Option('o', "output", HelpText = "Location to store the generated [redacted] files
    ↪ ")]
11 public string OutputPath { get; set; }
```

The overall logic of the command is captured in its Execute() method:

```
1  public void Execute()
2  {
3      // ... Initialization and input validation
4
5      var report = GetReport(ReportsApi, ReportId);
6
7      // Go through the report and convert CE objects to [redacted] objects
8      List<[redacted]> ceObjects = new();
9      foreach (var ceData in
10         from result in report.DesignRuleResults
11         from subproperty in result.Checks
12         where subproperty.Result is DesignRuleCheck.ResultEnum.Fail
13         select subproperty.Counterexample)
14     {
15         ceObjects.AddRange(ConvertCeData(ceData.VisualisationData));
16     }
17
18     // Use [redacted] to write [redacted] objects to files
19     [redacted] [redacted] = new(OutputPath, new XmlMetadata(Path.
    ↪ GetFullPath("[redacted]_Metadata/[redacted]+CE.xml", OutputPath)));
20     [redacted].LoadData();
21     foreach (var ceObject in ceObjects)
22     {
23         [redacted].CreateItem(ceObject);
24     }
25     [redacted].Save();
26
27     Environment.ExitCode = (int)ExitCode.Success;
28     return;
29
30     // ... Local function definitions
31 }
```

The first two steps are implemented as local functions. First, the GetReport() function retrieves a report from the server similar to the existing getreport command, with additional error checking:

```
1  static Report GetReport(IReportsApi api, string reportId)
2  {
```

```

3     try
4     {
5         var apiResponse = api.GetReportWithHttpInfo(reportId).Data;
6         return apiResponse switch
7         {
8             SingularBody s => s.Payload switch
9             {
10                 Report r => r,
11                 _ => throw new ██████████ ReportException(
12                     $"Payload of API response isn't a report: got {apiResponse.
13                         ↳ ObjectType}")
14             },
15             _ => throw new ██████████ ReportException($"Unexpected API response
16                 ↳ structure: {apiResponse}")
17         };
18     }
19     catch (ApiException e)
20     {
21         Log.Error("Failed to get report: " + e.Message, string.Empty,
22             ↳ LogClient.██████████);
23         Environment.ExitCode = (int)ExitCode.GetReportUnsuccessful;
24         throw;
25     }
26     catch (██████████ ReportException e)
27     {
28         Log.Error(e.Message, string.Empty, LogClient.██████████);
29         Environment.ExitCode = (int)ExitCode.GetReportUnsuccessful;
30         throw;
31     }
32 }

```

The counterexample data contained in the report are of the data types in the `RESTModel.Model` namespace which we generated from the OpenAPI schema, and are the same types used by the server to construct the report. The local function `ConvertCeData()` converts this data into `██████████` objects, each of which represents a row in a `██████████` file. In the style we introduced in example 2, we convert each type of visualisation object into a read-only collection of `██████████` then combine them into a single list.

```

1 static IReadOnlyCollection<██████████> ConvertCeData(VisualisationData
2     ↳ visualisationData)
3 {
4     IReadOnlyCollection<██████████> ██████████ = visualisationData.Nodes.Select(
5         ↳ ConvertCeNode).ToList();
6     IReadOnlyCollection<██████████> ██████████ = visualisationData.Portions.
7         ↳ Select(ConvertCePortion).ToList();
8 }

```

```

5  IReadOnlyCollection<[redacted]> [redacted] = visualisationData.Paths.Select(
    ↪ ConvertCePath).ToList();
6  [redacted] [redacted] = ConvertCeViolation(visualisationData.Metadata);
7
8  return new List<[redacted]>()
9      .Concat([redacted])
10     .Concat([redacted])
11     .Concat([redacted])
12     .Append([redacted])
13     .ToList();
14
15  // ... Local function definitions
16 }

```

Here, The `Select()` extension method [46] has similar semantics to the higher-order function *map* commonly found in functional programming languages — it applies a transformation function to each element of a collection, producing a collection of the transformed elements. In our case, these transformation functions are defined locally within `ConvertCeData()`. For example, the following function converts a single `CeNode` to its corresponding `[redacted]`:

```

1  static [redacted] ConvertCeNode(CeNode node)
2  {
3      var [redacted] = new [redacted] ([redacted].CeNode.TypeName);
4      [redacted].AddAttribute([redacted].CeNode.NodeId, node.NodeId);
5      [redacted].AddAttribute([redacted].CeNode.Track, node.Track);
6      [redacted].AddAttributeAsDecimal([redacted].CeNode.Distance, node.
    ↪ Distance);
7      [redacted].AddAttribute([redacted].CeNode.Direction, node.Direction);
8      // Unspecified visibility is treated as hidden
9      [redacted].AddAttribute([redacted].CeNode.Highlight, (node.Highlight ??
    ↪ false) ? "TRUE" : "FALSE");
10
11     return [redacted];
12 }

```

The final step is to write the `[redacted]` to `[redacted]` using existing functions in `[redacted]`. The `[redacted]+CE.xml` file is our extended schema for `[redacted]`, which contains the data types we added for representing counterexamples.

```

1  [redacted] [redacted] = new(OutputPath, new XmlMetadata(Path.
    ↪ GetFullPath("[redacted].Metadata/[redacted]+CE.xml", OutputPath)));
2  [redacted].LoadData();
3  foreach (var ceObject in ceObjects)
4  {
5      [redacted].CreateItem(ceObject);
6  }

```

7 | [redacted].Save();

Counterexample data is now stored in .[redacted] alongside the original scheme plan data, ready to be read by [redacted]. The next chapter will describe how we modified [redacted] to display this data.

Chapter 9

Implementing the Visualisations in [redacted]

Contents

9.1	Modifying the [redacted] Schema	93
9.2	Preparatory Work on the Codebase	97
9.3	Defining the Appearance of Counterexample Elements	102

The UK [redacted] ([redacted]) editor is a program that can create and modify railway infrastructure data in the [redacted] format [83] and can perform various tasks, including representing the data in a graphical form in its “Track Layout” view and interacting with the [redacted] [85].

[redacted] makes use of the [redacted] library to deserialise the [redacted] data into the *Track-Distance Model* (TDM, see Chapter 3). In [redacted] development, the TDM is also known as the “Railway Model” (RM).

9. Implementing the Visualisations in [redacted]



Figure 9.1: The transformations between data models involved in the process of graphically displaying a set of [redacted] data in [redacted].

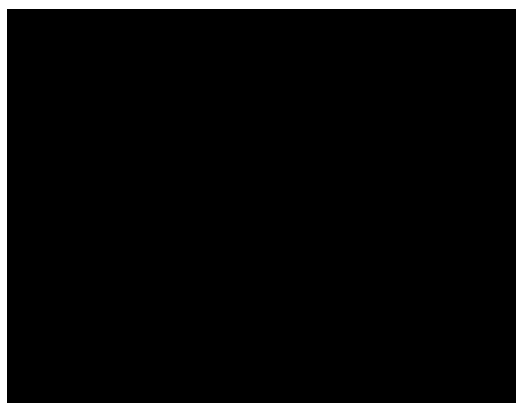


Figure 9.2: The [redacted] objects, each representing a row in a [redacted] file.

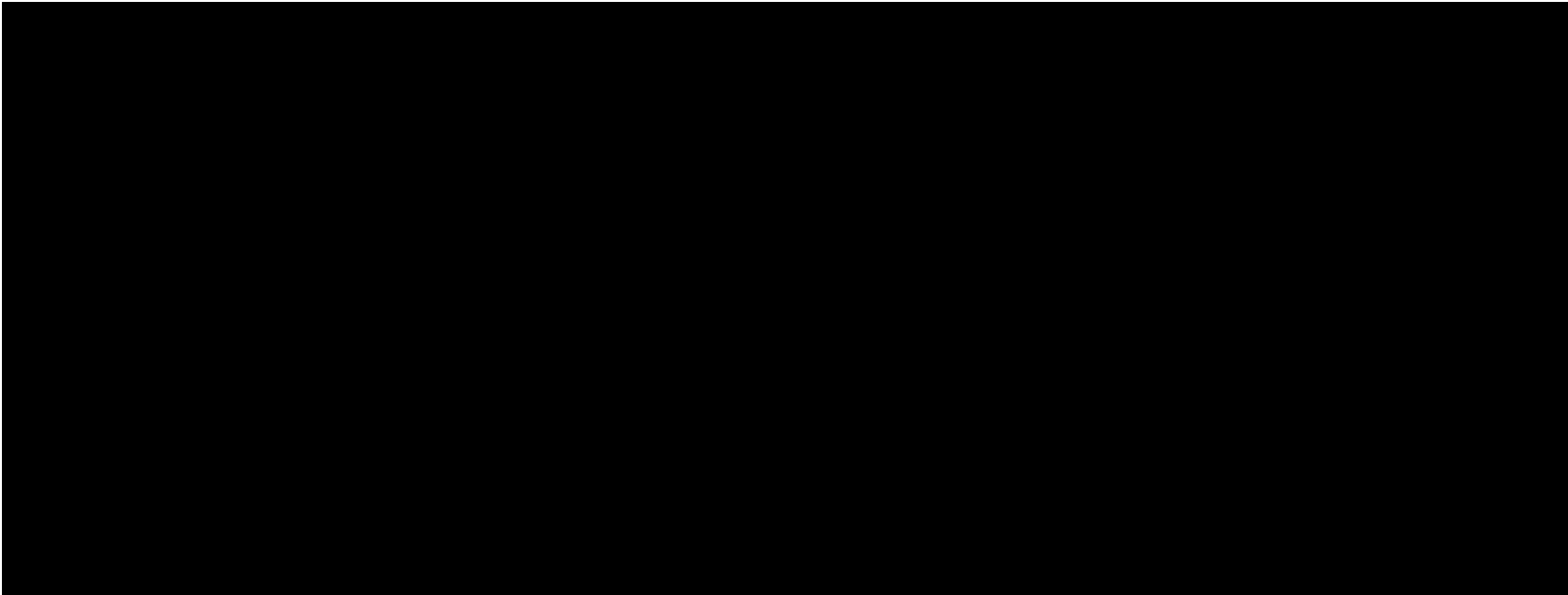


Figure 9.3: Attributes of a [redacted] object that represents a track.

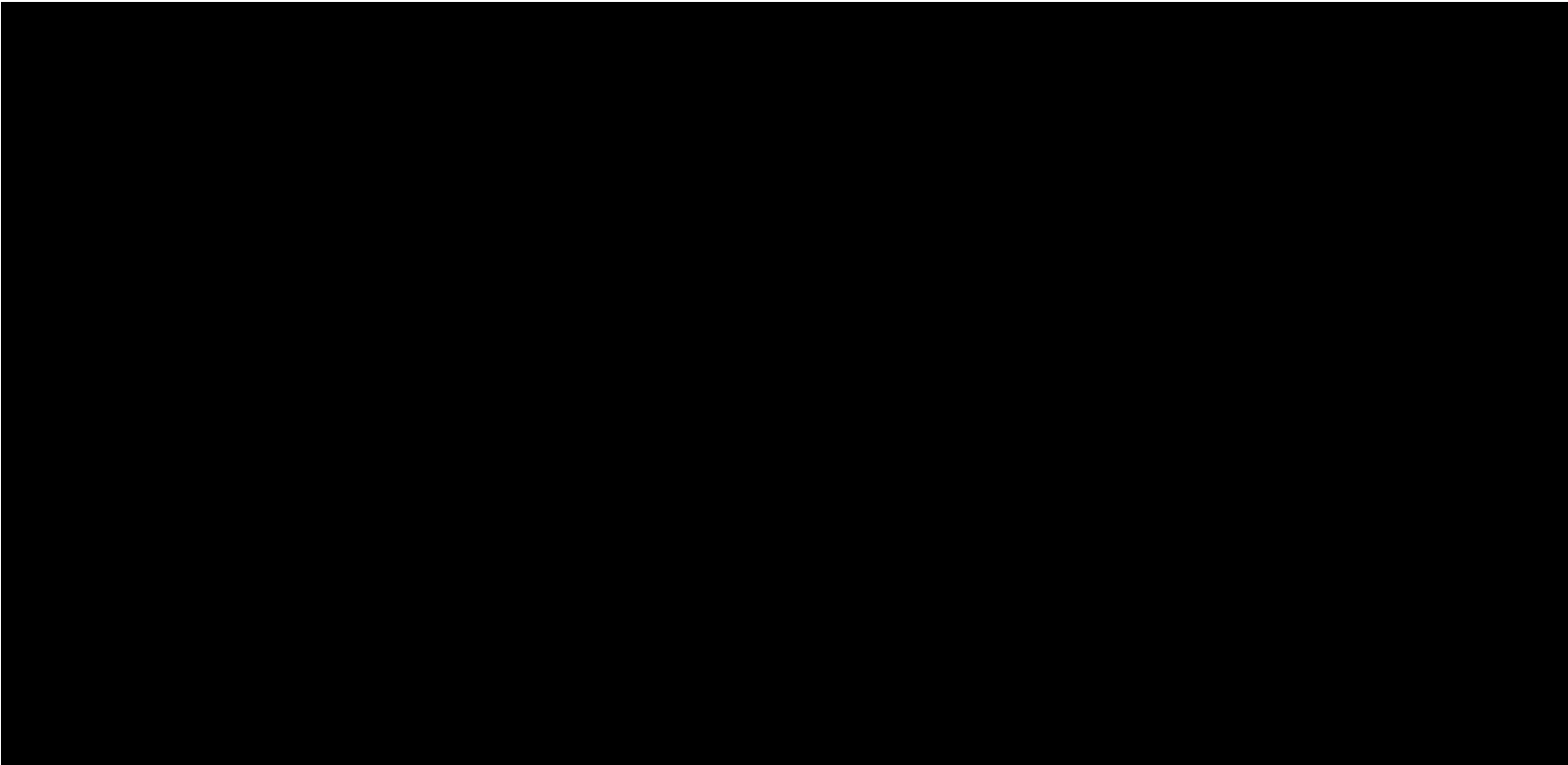


Figure 9.4: Data parsed from  files shown as tables in 

Figure 9.1 illustrates the data transformations involved in graphically displaying a set of scheme plan data. The first step is to parse the [redacted] files into a data model that directly represents the tables, the [redacted]. In this model, each row of the table is represented as an object of type [redacted], as shown in Figure 9.2. Each column of the table is represented as an *attribute* of the [redacted]. Figure 9.3 shows attributes parsed from the first *track*^T in the [redacted] file in Listing 5.2.

Since [redacted] are simply deserialised rows of TSV data, [redacted] can directly represent elements of each object type as a table, as shown in Figure 9.4.

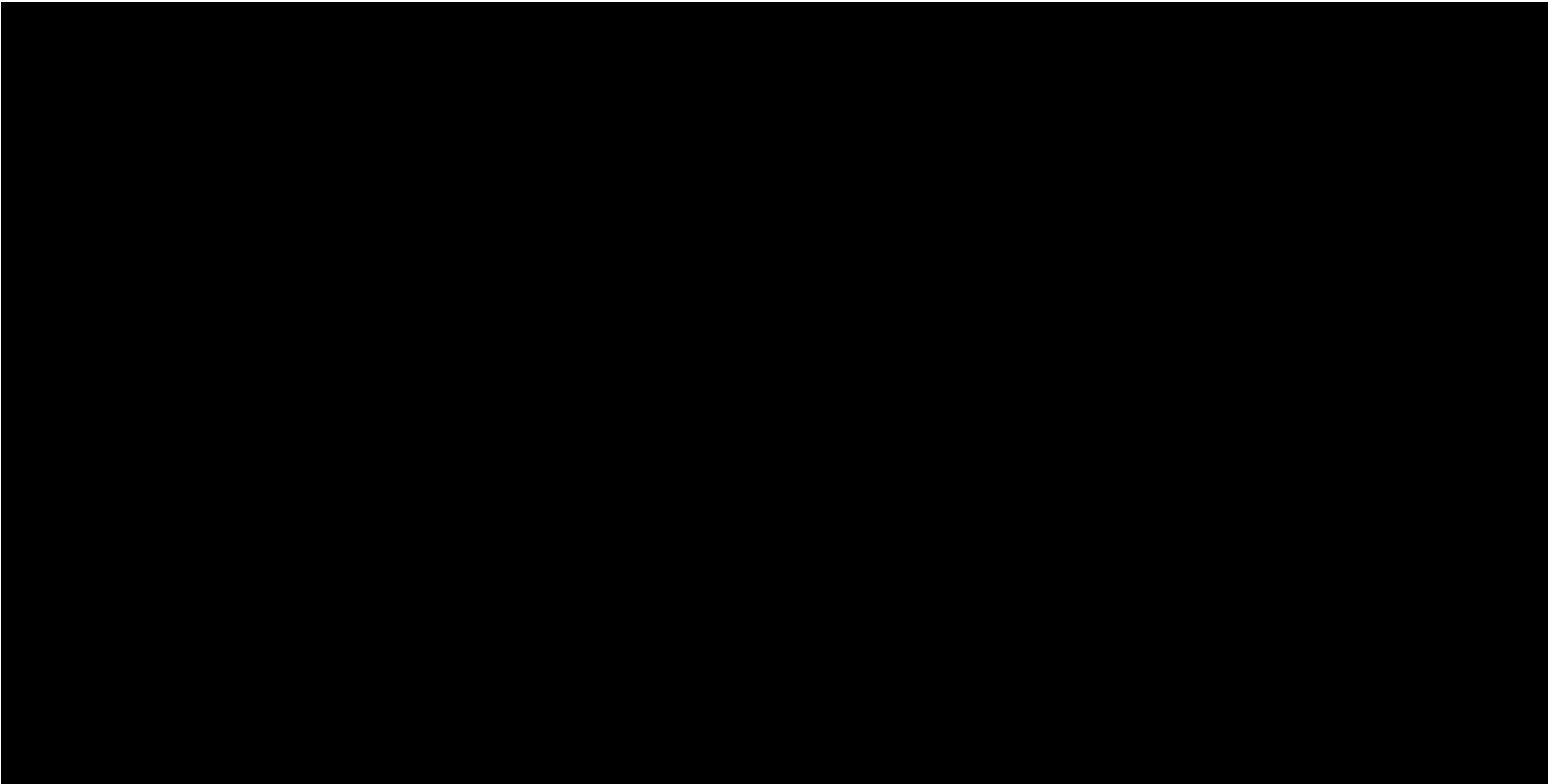


Figure 9.5: The RMTTrack representing the first track in Listing 5.2.

Next, data is transformed into a model known as the `RailwayModel` (RM). In this model and all subsequent models, we have distinct types for different kinds of model elements; however, not every type of object in the scheme plan has its own type. Whether or not an element type has a corresponding data type is determined by its data members. For example, balises, AWS magnets, and marker boards are all encoded as `RMTDPosition` objects, since they are all positions defined with a track and distance, and contain the same data members. Each object in the `RailwayModel` includes a reference to its corresponding [redacted], which provides a means of looking up its original element type and attributes. Figure 9.5 shows the RM object that corresponds to the track in our example.

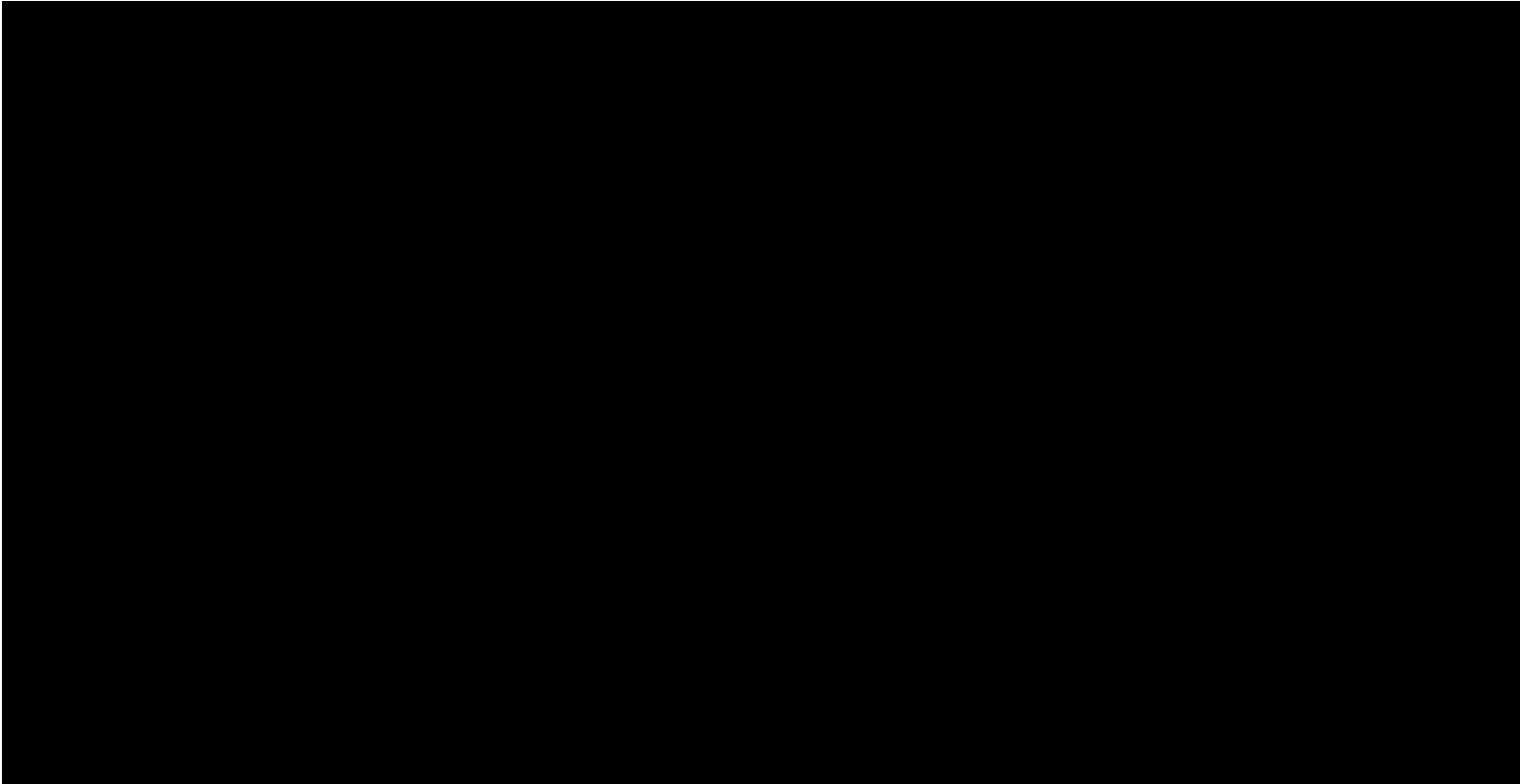


Figure 9.6: A segment of the example track represented in models used for rendering the track layout diagram.

From here, [redacted] performs further model transformations depending on what it is asked to do — if the user chooses to display the scheme plan graphically, it transforms the RM objects into the ViewData model and then the Symbols model. Both of these models represent the elements in a way that reflects how they are graphically rendered, adding information such as their coordinates on the canvas. For example, tracks are split up into segments (TrackLines) to represent their appearance on the diagram, as shown in Figure 9.6. Elements of these data models allow us to go all the way back to the [redacted] they represent: in the case of a track segment, the Symbol version stores a reference to the ViewData version, which in turn stores the ID of the track used in RailwayModel.

The user interface of [redacted] is built with the Windows Presentation Foundation (WPF) framework [60]. The appearance of UI elements — which include shapes representing infrastructure objects in the “Track Layout” view — are defined using XAML files [61] containing *styles* and *templates* [58]. For the “Track Layout” view, the following files are used:

- SymbolsTemplate.xaml, containing *styles* that can be shared between infrastructure objects.
- DefaultSymbolTemplate.xaml, containing *templates* that make use of the *styles* to define the appearance of each type of infrastructure object.

We will refer to both of these files collectively as the “symbol template” files.

Based on this architecture, in order to visualise counterexamples we need to do the following:

1. Extend the [redacted] format with our own data types that represent components of counterexamples.
2. Choose the data types used to represent counterexample elements in each of the data models.
3. Extend the symbol template files to define appearances of counterexample elements.

9.1 Modifying the [redacted] Schema

[redacted] is an XML file that serves as the schema for [redacted], so it is the central starting point for incorporating counterexample data into [redacted]. It has the following overall structure:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <[redacted] xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
  ↳ noNamespaceSchemaLocation="[redacted].xsd">
3   <Metadata>
4     <!-- Version, copyright, etc -->
5   </Metadata>
6   <ObjectGroups>
7     <!-- ... -->
8   </ObjectGroups>

```

```
9      <DataTypes>
10          <!-- ... -->
11      </DataTypes>
12      <ObjectTypes>
13          <!-- ... -->
14      </ObjectTypes>
15      <TypeClasses>
16          <!-- ... -->
17      </TypeClasses>
18      <Relationships>
19          <!-- ... -->
20      </Relationships>
21 </[redacted]>
```

Of note here are the following elements:

- **DataTypes** form the basis of the type system, defining valid values or ranges of fundamental data elements (e.g. acceptable values for distances).

This listing has been removed for intellectual property reasons.

- **ObjectTypes** contain definitions of data structures that make use of the **DataTypes** to represent infrastructure elements and more abstract concepts (e.g. routes).

This listing has been removed for intellectual property reasons.

This listing has been removed for intellectual property reasons.

- ObjectGroups categorize the ObjectTypes and are used by tools to decide how to handle each object type. In the case of [redacted], the ObjectGroup an object type belongs to decides how it is drawn in the “Track Layout” view.

This listing has been removed for intellectual property reasons.

Below are our definitions of counterexample object types. Note that although we have created a ObjectGroup called “Counterexample Objects”, not all CE object types belong to it: CENode belongs to “Position Objects” and CEPPath belongs to “*Path*^T Objects” so that they can take advantage of the rendering logic in [redacted] for similar objects, eliminating the need to reimplement the complex calculation of cartesian coordinates and scaling ourselves.

```
<ObjectType Name="CENode" [redacted] FilePrefix="ce_nodes" Description="Overlay
  ↳ node for building the counterexample, may be optionally highlighted."
  ↳ ObjectGroup="Position Objects">
  <ObjectFields>
    <ObjectField Name="node_id" Type="Identifier"/>
    <ObjectField Name="track_id" Type="Track"
      Description="The (one and only) track the node is
        ↳ associated with" />
    <ObjectField Name="distance" Type="Distance" Description="Distance of
      ↳ the node measured from the datum of the track." />
    <ObjectField Name="direction_id" Type="Direction" Description="
      ↳ Direction the counterexample applies to." />
    <ObjectField Name="highlight" Type="Boolean"
      Description="Decides whether or not the node should be
        ↳ highlighted; if FALSE, the node is invisible." />
  </ObjectFields>
  <IdAttributes>
    <IdAttribute>node_id</IdAttribute>
  </IdAttributes>
</ObjectType>
```

```

<ObjectType Name="CEPortion" FilePrefix="ce_portions" Description="A
    ↪ part of a track to be highlighted." ObjectGroup="Counterexample
    ↪ Objects">
  <ObjectFields>
    <ObjectField Name="path_id" Type="CEPath" Description="Path the
        ↪ portion belongs to." />
    <ObjectField Name="portion_id" Type="Identifier" />
    <ObjectField Name="track_id" Type="Track" Description="The track the
        ↪ portion is taken from." />
    <ObjectField Name="node0_id" Type="CENode" Description="Node the
        ↪ portion starts from." />
    <ObjectField Name="node1_id" Type="CENode" Description="Node the
        ↪ portion ends at." />
  </ObjectFields>
  <IdAttributes>
    <IdAttribute>path_id</IdAttribute>
    <IdAttribute>portion_id</IdAttribute>
  </IdAttributes>
</ObjectType>

<ObjectType Name="CEPath" FilePrefix="ce_paths" Description="A set of
    ↪ portions to be highlighted." ObjectGroup="Path Objects">
  <ObjectFields>
    <ObjectField Name="path_id" Type="Identifier" />
    <ObjectField Name="violation_id" Type="CEViolation"/>
    <ObjectField Name="entrance_node_id" Type="CENode"/>
    <ObjectField Name="exit_node_id" Type="CENode"/>
  </ObjectFields>
  <IdAttributes>
    <IdAttribute>path_id</IdAttribute>
  </IdAttributes>
  <PathDetails EntrancePositionItem="entrance_node_id" ExitPositionItem="
    ↪ exit_node_id"/>
</ObjectType>

<ObjectType Name="CEViolation" FilePrefix="ce_violations" Description
    ↪ ="A group of paths and metadata describing a counterexample."
    ↪ ObjectGroup="Counterexample Objects">
  <ObjectFields>
    <ObjectField Name="violation_id" Type="Identifier"/>
    <ObjectField Name="rule_id" Type="string"/>
    <ObjectField Name="description" Type="string"/>
  </ObjectFields>
  <IdAttributes>

```

```
<IdAttribute>violation_id</IdAttribute>
</IdAttributes>
</ObjectType>
```

9.2 Preparatory Work on the Codebase

[redacted] and [redacted] are both large projects with multiple data models, so to work out how to correctly implement our additions we first needed to refactor the existing code to make the logic clearer. As an example, in [redacted] the `RailwayModel.RmTdPath` class implements many operations related to *path* objects, which each represents a section of track between two defined *positions* (locations with a distance and optionally a direction). When implementing the `CEPath` object type, we encountered bugs where our data does not get transformed correctly, which were caused by our data types missing some fields that were required by the `Calculate()` method.

Listing 9.1: Outline of the `Calculate()` method before refactoring.

```
1 This listing has been removed for intellectual
2 property reasons.
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```
24 This listing has been removed for intellectual property
25 reasons.
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
```

The original method is shown in Listing 9.1. It is unclear what the method does from just reading, so we observe its behaviour using a debugger, and refactor it while preserving the existing semantics:

- The condition on the `if` statement on line 5 allows execution to continue (skipping the body of the `if`) even if the state is invalid, which makes it unclear what the method will do in that case; we make the program flow easier to follow by terminating the method early with a guard clause in an invalid state.
- There is only one `return` statement, but the return value can be affected by many factors in the nested branching structure — to determine the result of a given method call, one would need to trace through the entire method; we structure the branching such that each invalid condition will cause the method to exit early (i.e. the program graph for the method becomes a tree-like shape), which also allows us to use literal return values in most cases.
- The computation of path portions is done in 3 places, and the temporary variables of the first computation (line 15) is captured when this is repeated on lines 26 and 32; we isolate this procedure into a local function so that its state does not have unintended effects.

Listing 9.2: Outline of the `Calculate()` method after refactoring.

```
1 This listing has been removed for intellectual property
2 reasons.
3
```

4 This listing has been removed for intellectual property
5 reasons.

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

```
44 This listing has been removed for intellectual property
    reasons.
45
46
47
48
49
50
51
52
53
```

These changes resulted in the refactored method shown in Listing 9.2. Note that our modifications are more conservative compared to in the [REDACTED] project (see example 2), because we want to extend the functionality of [redacted] and [REDACTED] using new data and minimal code changes, in order to avoid breaking backward compatibility whenever possible.

Listing 9.3: Outline of the AddToPosition() method before refactoring.

```
1 This listing has been removed for intellectual property
  reasons.
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```

24 This listing has been removed for intellectual property
25 reasons.
26
27

```

Another example of our refactoring work is the `ItemView.AddToPositions()` method in `ItemView` which is a part of the transformation process from `RailwayModel` to `ViewData`. In the original version of this method — shown in Listing 9.3 — the value of the variable `positionViewLocation` is returned at the end (line 26) but it is declared a long time earlier (line 3), before the complex branching logic. After analysing the behaviour of the branching logic, we found that this variable is not modified before line 23, so we decided to move the declaration to just before the variable is used. In addition, we reduce the level of nesting by turning the `if` statement on line 4 into a guard clause, as we did in the previous example.

Listing 9.4: Outline of the `AddToPosition()` method after refactoring.

```

1 This listing has been removed for intellectual property
2 reasons.
3
4
5
6
7
8
9
10
11
12
13

```

The result of refactoring the `AddToPositions()` method is shown in Listing 9.4. Because this method operates largely via side effects and has several referential dependencies — neither of which can be inferred from observing the parameters passed and values returned, we also add the following method documentation:

This section has been removed for intellectual property reasons.

This section has been removed for intellectual property reasons.

9.3 Defining the Appearance of Counterexample Elements

A CENode is shown as a orange circle at the appropriate location on the scheme plan. It shares the CanvasStyle with many other position objects, reusing the definitions there to inherit their behaviour in response to events such as scaling and rotation (when a track is shown at an angle on the plan).

```
<DataTemplate x:Key="CENodeTemplate">
  <Canvas Style="{StaticResource CanvasStyle}">
    <Ellipse Canvas.Top="-2" Canvas.Left="-2" Height="4" Width="4" Style
      ↪="{StaticResource CounterexampleShapeStyle}" Fill="DarkOrange
      ↪"/>
  </Canvas>
</DataTemplate>
```

We also define a common style shared by counterexample objects, including ones we might implement in the future (such as nodes with different colours or shapes).

```
<Style x:Key="CounterexampleShapeStyle" TargetType="Shape">
  <Setter Property="StrokeThickness" Value="0.1"></Setter>
  <Setter Property="Opacity" Value="0.7"/>
</Style>
```

Listing 9.5: The XAML template for CEPATH.

```
1 <DataTemplate x:Key="CEPathTemplate">
2   <Canvas
3     Name="pathCanvas"
4     Margin="{Binding AreaDisplaySlot, Converter={StaticResource
5       ↪ PathMarginCoverter}}"
6     ToolTip="{Binding Path=Name}">
7     <ItemsControl
8       Name="itmControl"
9       HorizontalAlignment="Left"
10      VerticalAlignment="Top"
11      ItemsSource="{Binding Path=AreaItem.TrackList, Mode=TwoWay}"
```

```

11     ScrollViewer.HorizontalScrollBarVisibility="Auto">
12     <!-- ... -->
13     <ItemsControl.ItemTemplate>
14         <DataTemplate>
15             <Line
16                 X1="{Binding StartX}"
17                 X2="{Binding EndX}"
18                 Y1="{Binding StartY}"
19                 Y2="{Binding EndY}" Fill="DarkOrange"/>
20                 Style="{StaticResource LineAreaPathStyle}"
21             </DataTemplate>
22         </ItemsControl.ItemTemplate>
23     </ItemsControl>
24 </Canvas>
25 </DataTemplate>

```

By comparison, the implementation of `CEPath` (Listing 9.5) is more complex, because its appearance changes based on properties of the track(s) it is drawn over. WPF uses *data binding* to connect UI elements with underlying data [45]. This is useful when display properties directly correspond to data properties, such as the start and end coordinates defined in lines 15-19; however, for display properties that require further computation from the underlying data, *converter* classes are used. `CEPath` uses the existing `LineAreaPathStyle` to inherit the existing logic for path objects defined there, such as the calculation of stroke thickness depending on the zoom level implemented in the `AreaPathThicknessCalculator` class:

```

<Setter
    Property="StrokeThickness"
    Value="{
        Binding ElementName=itmControl,
        Path=DataContext.Zoom,
        Converter={StaticResource AreaPathThicknessCalculator}
    }"
/>

```

While `CENode` is implemented as a position object and `CEPath` is implemented as a path object, we still want to use our own styling in `CEPath` instead of the default ones for their types in the Symbols model (`Position` and `Area`, respectively). This means we need to modify the *converter* classes which select templates for Symbols objects based on their underlying data.

Listing 9.6: Updated implementation of `SymbolTemplateSelector`.

```

1 This listing has been removed for intellectual property
2 reasons.
3
4
5

```

```
6 This listing has been removed for intellectual property
7 reasons.
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

The overall matching logic for all object types is implemented in the `SymbolTemplateSelector` class. We incorporate the counterexample objects into the matching logic (shown in Listing 9.6) using pattern matching [56], and in the following order:

1. All `Area` objects other than `CEPath` shall use the existing generic `AreaTemplate`.
2. For other object types, if there is an existing template for their underlying `ItemType`, use that template (this includes `CENode` and `CEPath`).
3. For position objects without a dedicated template, use one of the generic position templates based on their directionality.
4. For all other objects, use the fallback `NoTemplate`.

Similarly, the `AreaPathStrokeCalculator` selects colours of `Area` objects based on their `ItemType` property. We refactored the existing if-else based implementation into one using pattern matching, and separated the mapping from item types to colours into a static local function, where we add the definition for `CEPath`:

```
This listing has been removed for intellectual property
reasons.
```

This listing has been removed for intellectual property reasons.

With these changes to , we are now able to display a counterexample produced by the  as an overlay on the scheme plan. The next chapter will demonstrate an end-to-end workflow of verifying a set of scheme plan data in  and visualising the counterexample from the result.

Chapter 10

An Extended Example of Counterexample Visualisation

In this chapter, we demonstrate the entire process of verifying a scheme plan against a design rule and visualising a counterexample. We show excerpts of files involved in this example — the full contents of the files are in Appendix E.

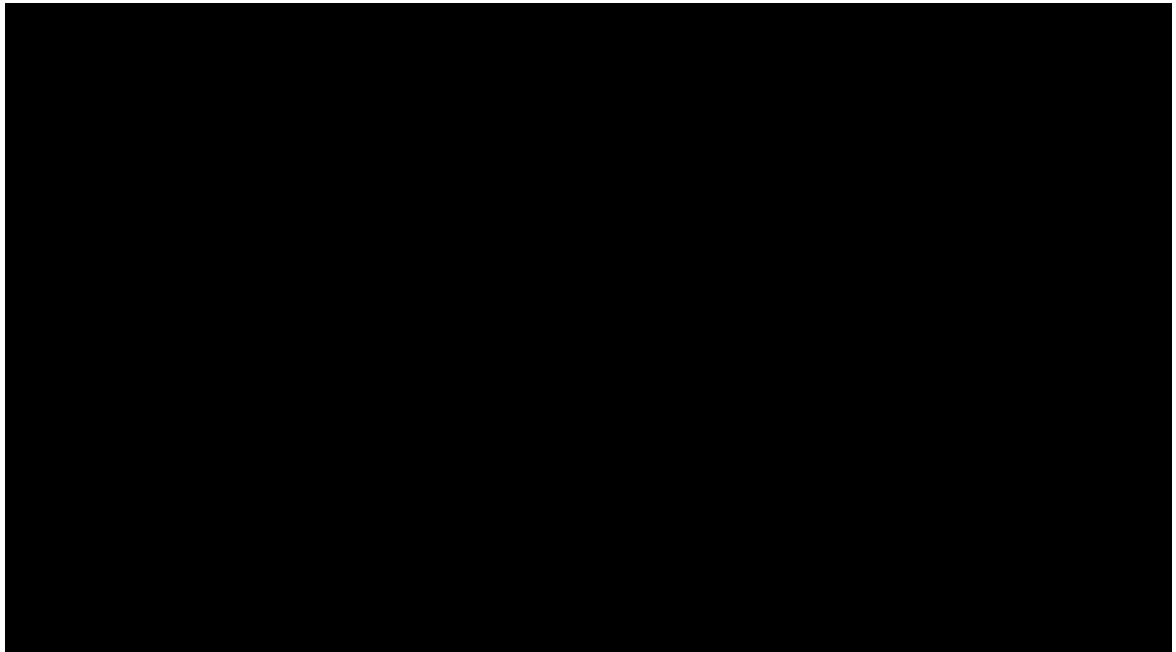


Figure 10.1: The scheme plan used in our extended example, as displayed in

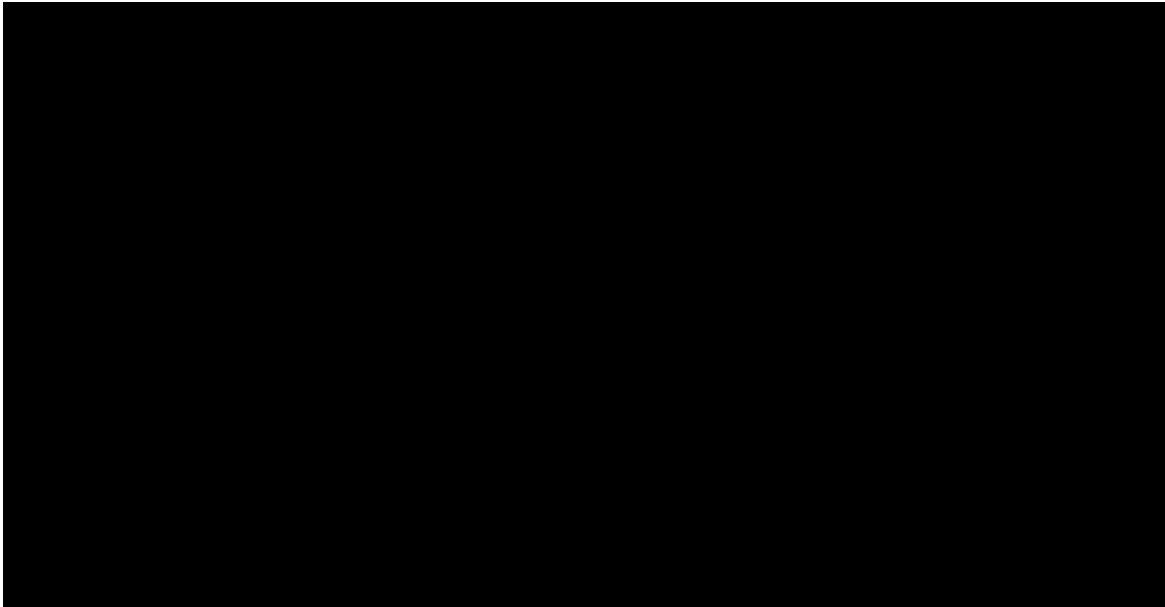


Figure 10.2: Infrastructure data from [redacted] files displayed in [redacted] as tables.

We start by opening a set of `[redacted]` files in [redacted] that contain infrastructure data for a scheme plan. The scheme plan can be displayed in graphical form in the “Track Layout” tab (Figure 10.1), and the underlying [redacted] data can also be shown as tables in the “Editor” tab (Figure 10.2).

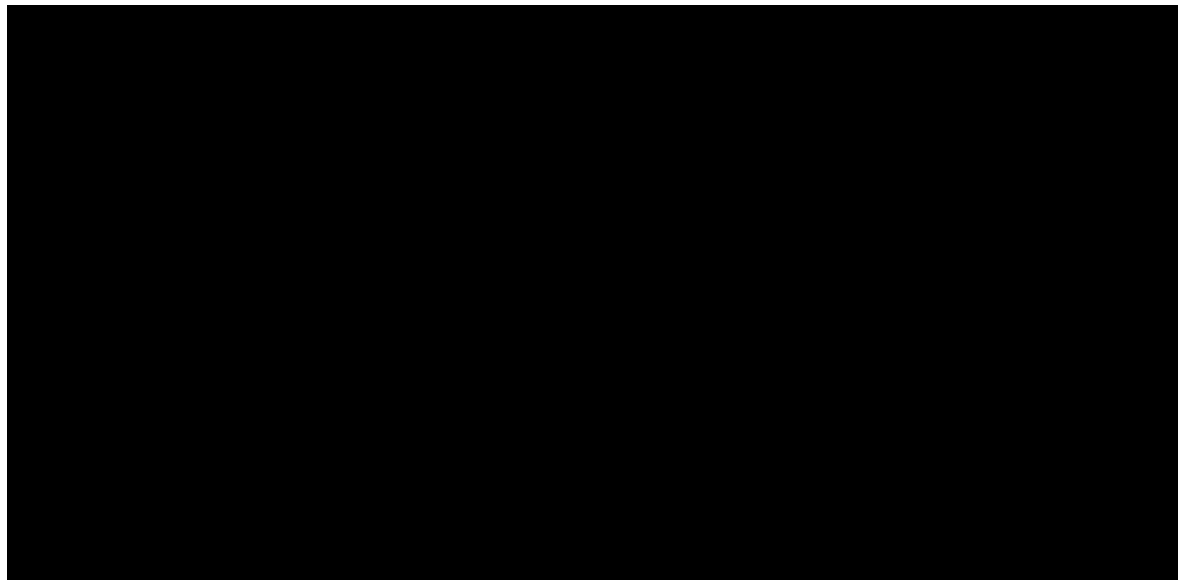


Figure 10.3: The “[redacted]” tab of [redacted] prior to starting verification.

To verify the scheme plan, we switch to the “[redacted]” tab (Figure 10.3) and select a design rule to be checked. For this example, we use a slightly modified version of rule *BG-03-1a*, which specifies the minimum spacing between a balise and point toes. We change

the value of the minimum spacing to 20.0m, so that the visualisation of the counterexample is easier to see (it would otherwise be too small). The design rule is encoded in an XML file that is sent to the [REDACTED] along with the [REDACTED] data:

```
1 <drl:DesignRule
2   xmlns:pro="..." xmlns:drl="..." xmlns:xsi="..."
3   id="BG-03-1a-demo">
4   <pro:Description>
5     Design placement of balises should avoid points and crossings. Designed
6     ↪ spacing shall be constrained by:
7
8     1a. This design rule checks between balise and PointsNode.
9   </pro:Description>
10  <pro:Forall>
11    <!-- The constraint expressed as a first-order logic formula -->
12  </pro:Forall>
13  <drl:Rationale>
14    <!-- Textual justification for the constraint -->
15  </drl:Rationale>
16  <drl:SafetyCritical>
17    true
18  </drl:SafetyCritical>
19  <drl:ErrorMessageTemplate>
20    Spacing between balise {0} and point toes of {1} is less than 20.0m
21  </drl:ErrorMessageTemplate>
22 </drl:DesignRule>
```

The full content of this design rule file is shown in Listing E.1

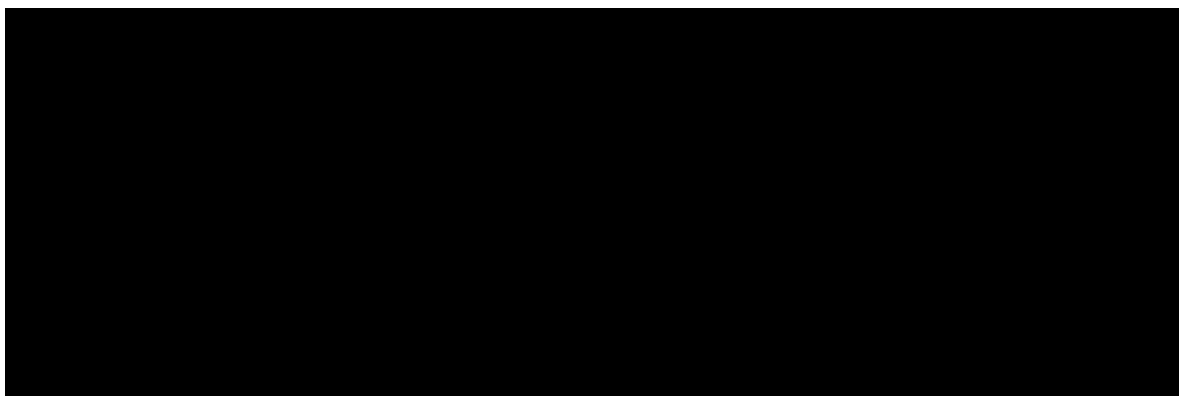


Figure 10.4: The “ [REDACTED] ” tab of [REDACTED] after starting verification.

When we click the “Verify” button, [REDACTED] sends a verify command to the [REDACTED] client, which in turn sends a REST API call to the server to start the verification. The server creates

a new verification job and responds with the ID of the job. The client writes this information to a temporary XML file, which [redacted] uses to show the job status (Figure 10.4).

While the verification runs, we can use the “Refresh” button to check its progress, which causes [redacted] to send a `getjob` command to the [redacted] client, and read the job status from the temporary XML file produced.

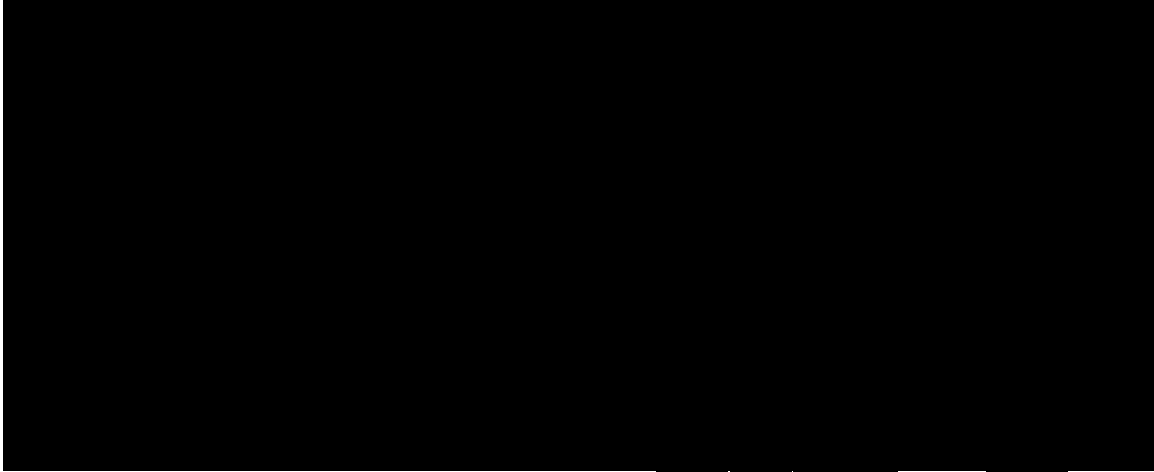


Figure 10.5: Verification results displayed in the “[redacted]” tab of [redacted]

Eventually, the [redacted] indicates that it has finished the verification, which causes [redacted] to retrieve verification results using the `getreport` command. The [redacted] client contacts the server using the REST API, and writes the report data into an XML file, shown in Listing E.2. The overall outcome is shown alongside the result of each subproperty (Figure 10.5). In this case, the violating subproperty produced the following unsat core when checking with Z3:

```
1 (distance_to_distancebound_2 def_edge_T1_node_4_3_PointsNode5
   ↪ def_pointsnode_PointsNode5 def_position_Balise_4_3 rule2)
```

The contents of the unsat core is converted to the following error message:

Spacing between balise 4_3 and point toes of PointsNode5 is less than 20.0m (Actual distance was 15m)

When we click the “Show Counterexample” button, [redacted] issues the `get[redacted]` to the [redacted] client. The counterexample data is written to additional [redacted] and visualised in the “Track Layout” tab (Figure 10.6). We can also inspect the additional data as tables in the “Editor” tab (Figure 10.7). The raw content of the generated [redacted] files are in listings E.3, E.4, E.5, and E.6.

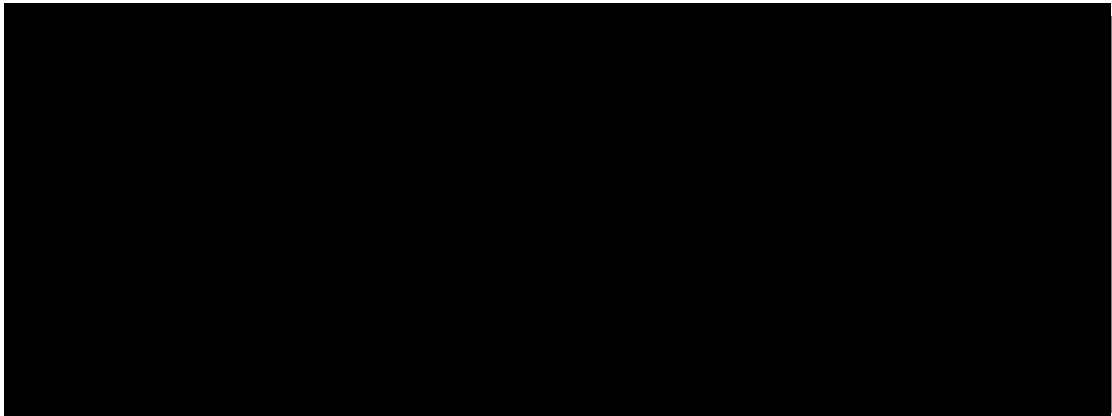


Figure 10.6: The counterexample displayed in graphical form.

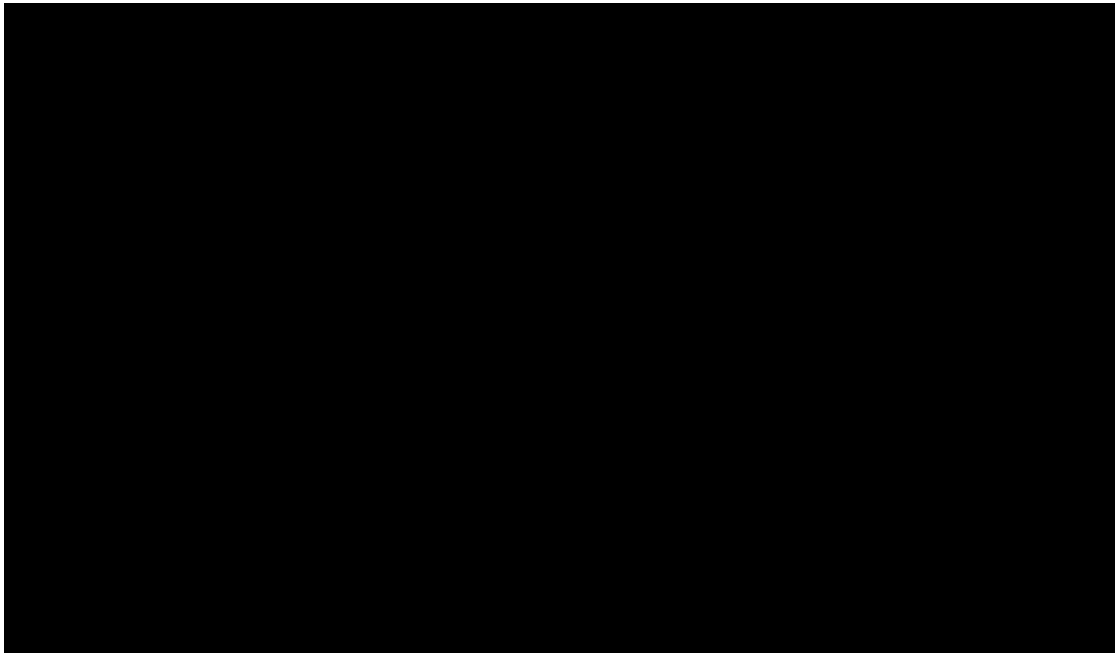


Figure 10.7: Counterexample data from our additional .[redacted] files, shown in tabular form.

The counterexample found by the toolchain reflects the violation of rule BG-03-1a (with our slight modifications to the distance values) on this scheme plan. The visualisation in [redacted] provides a graphical indication of the location of the violation to the railway engineer working on the scheme plan, and its presentation is according to the designs we produced following our focus group study. At this point, changing the distance of balise 4_3 to 780m, and then re-running the verification with the [redacted] would result in a *Pass* outcome. Thus, we have shown that counterexample visualisation supports an engineer's workflow in finding and correcting mistakes in scheme plan data.

Part III

Conclusion

Chapter 11

Summary

In this thesis, we presented an extensive software development project extending the [redacted] scheme plan verification toolchain with the functionality to visualise counterexamples. It involved the following activities:

- A focus group study with users of the toolchain to co-develop requirements on the presentation of counterexamples (Chapter 6).
- Development of a software architecture (Chapter 7) covering the components, interfaces and communications required to facilitate counterexample visualisation. This consisted of:
 - Extracting counterexample information from the proof tool (in the [redacted] server, Chapter 8).
 - Defining an extended DSL for visualisation data ([redacted], Chapter 9 and its counterpart in the [redacted]'s communications, Chapter 8).
 - Transforming the extracted information into the extended DSL (in the [redacted] server and client, Chapter 8).
 - Extending a domain-specific data authoring program with the ability to display visual representation of counterexamples using information in the extended DSL ([redacted], Chapter 9).
- Implementation of the functionality following this architecture, through both C# programming and extensions to the data formats involved, demonstrating that our concept works.

We demonstrated the working of this reverse data transformation in an extended example (Chapter 10), using a scheme plan of industrial size. Due to the need to align development work in R&D with project delivery, our work has not yet been merged into the main (release) branches of the [redacted] and [redacted] repositories; however, once an opportunity to integrate our work into the main toolchain arises, we believe our impact will be represented in the

reduction of time spent by engineers to refine their scheme plan designs following discovery of errors.

Our learning points from the experience of this project include:

- Formal method tools that interface with domain-specific software need user input for interaction design, so that the resulting product fits well within the industrial user's workflow.
- Software production in an industrial context can be challenging, due to the need to maintain backward compatibility with existing components and formats.
- Prototyping for the added features might have been easier in a purely academic environment, as there would be much less refactoring overhead from introducing interfaces where the software wasn't previously designed to have such an interface (cf. our modifications to the [REDACTED] server in Chapter 8).

Chapter 12

Future Work

Contents

12.1	Near Term Improvements	117
12.2	The Next Project	118
12.3	Possible Future Development in Counterexample Visualisation	119

In this chapter, we describe possible future work that could follow from this project.

12.1 Near Term Improvements

Given additional time, we would like to build on our proof-of-concept work and fully implement counterexamples for all five categories of design rules we identified in subsection 6.1.2.1. This would require us to solve the following problems:

- To correctly implement a `CEPath` consisting of multiple `CEPortions`, the code in the [redacted] server that extracts counterexample data from unsat cores would need to calculate the ordering of NEM edges (which correspond to `CEPortions`) so that it can find the entrance and exit of the `CEPath`.
- To represent violation paths with multiple branches (e.g. for a design rule in the “prohibited placement” category), we need to introduce a new counterexamples object type in [redacted] in the *Area Objects* group.
- Extract information to populate the `RelevantElements` component of counterexamples, to enrich the information presented to the user in [redacted] possibly allowing them to navigate to the elements involved in a violation.

In addition, we would like to allow the user to “reset” the Track Layout view of [redacted] to remove the counterexample objects or bring them back later. We envision this functionality to be realized in three aspects:

- A “Toggle Counterexamples” button to hide or show all counterexample objects at once (current [redacted] functionality allows toggling individual object types).
- A clear indication in the Track Layout view that the user is looking at a set of “post-check” [redacted] data that contains counterexamples from a specific check (possibly showing the report ID), so that after the user makes modifications to the scheme plan after verification, they do not misinterpret the counterexample objects shown as applying to the updated scheme plan.
- A “Remove Counterexamples” button to delete the `ce_*. [redacted]` which should also remove the “post-check” indicator. This allows the user to use the set of “clean” [redacted] data in some other tool for further work. This button should be implicitly pressed before every verification run.

Finally, we would like to extend the `get [redacted]` command we added to the [redacted] client with the ability to generate [redacted] files from a report previously retrieved from the [redacted] server and stored in an XML file. This would enable [redacted] to visualize counterexamples independently of the [redacted] server, perhaps some time after the verification was run.

Once these features have been implemented, we would like to hold a feedback session with the engineers, similar to the focus group session we ran to drive the initial design, so that we can validate our work and measure the value it brings to the [redacted] toolchain.

12.2 The Next Project

A project building upon our work could extend the counterexample visualisation toolchain in the following ways:

- Implement counterexamples for all design rules supported by the [redacted] not just the ones related to balise groups.
- While the [redacted] only allows verifying against one design rule at a time, [redacted] lets the user verify a scheme plan against multiple design rules in the same session, by submitting them to the [redacted] as separate verification jobs to be run sequentially. In this case the user may want to see counterexamples for all checked rules at the same time. Implementing this functionality would require further user consultation to come up with interaction designs that distinguish between different design rules.
- In a similar vein, it may be desirable to provide a user interface for filtering the counterexamples being displayed, based on the specific priorities of the user and their project with regard to modifications they would make to a scheme plan following a design rule violation.
- While the [redacted] can find direct violations of design rules, such violations are often indirectly caused by underlying errors that would only be discovered after fixing the immediate problem. Future development of the toolchain could use *causality* principles to help uncover root causes of violations [6]. This could help reduce the number

of verification-visualisation-redesign cycles required to make a scheme plan fully compliant.

12.3 Possible Future Development in Counterexample Visualisation

In the longer term, we think it would be beneficial to the general area of counterexample visualisation to develop a generic tool for data transformations similar to the ones we implemented. The tool would let the user choose their input (proof tool) and output (domain specific) languages — perhaps both specified in a standardized schema file format similar to OpenAPI and [redacted] — as well as a specification of the transformation.

Implementing this as a software library would enable integrating this functionality into more elaborate and specialized tool suites for industrial use like [redacted]; the tool would specify the languages and transformation, and provide them to the library as parameters.

Bibliography

- [1] ABO, R., AND VOISIN, L. Formal implementation of data validation for railway safety-related systems with OVADO. In *SEFM'13* (2013), LNCS 8368, Springer.
- [2] ABRIAL, J.-R. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.
- [3] BANERJEE, M., CAI, V., LAKSHMANAPPA, S., LAWRENCE, A., ROGGENBACH, M., SEISENBERGER, M., AND WERNER, T. A tool-chain for the verification of geographic scheme data. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification* (Cham, 2023), B. Milius, S. Collart-Dutilleul, and T. Lecomte, Eds., Springer Nature Switzerland, pp. 211–224.
- [4] BARRETT, C., SEBASTIANI, R., SESHIA, S., AND TINELLI, C. Satisfiability modulo theories. In *Handbook of Satisfiability, Second Edition*, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., vol. 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Feb. 2021, ch. 33, pp. 825–885.
- [5] BARRETT, C., STUMP, A., AND TINELLI, C. The SMT-LIB Standard: Version 2.6. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)* (2010, updated 2021), A. Gupta and D. Kroening, Eds.
- [6] BEER, I., BEN-DAVID, S., CHOCKLER, H., ORNI, A., AND TREFLER, R. J. Explaining counterexamples using causality. *Formal Methods Syst. Des.* 40, 1 (2012), 20–40.
- [7] BERGER, U., JAMES, P., LAWRENCE, A., ROGGENBACH, M., AND SEISENBERGER, M. Verification of the European Rail Traffic Management System in real-time Maude. *Science of Computer Programming* 154 (2018), 61–88.
- [8] BESMER, A., AND LIPFORD, H. Tagged photos: Concerns, perceptions, and protections. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI EA '09, Association for Computing Machinery, p. 4585–4590.
- [9] CAI, V., LAKSHMANAPPA, S., LAWRENCE, A., ROGGENBACH, M., AND SEISENBERGER, M. Counterexample visualisation for constraint checking in the railway domain, 2022. Abstract accepted for presentation at WADT '22, Aveiro, Portugal.

- [10] CENELEC. En 50128. *Railway applications-Communication, Signaling and processing systems-software for railway control and protection systems* (2011).
- [11] CHACON, S., AND STRAUB, B. *Pro git*. Springer Nature, 2014, ch. 7.11.
- [12] DEAVES, P. Railway codes and other data. <http://www.railwaycodes.org.uk/index.shtml>, 2023. [Online; accessed 2023-12-21].
- [13] DOCKER INC. Docker overview. <https://docs.docker.com/get-started/overview/>, 2023. [Online; accessed 2023-12-19].
- [14] ECMA INTERNATIONAL. ECMA-404 — The JSON Data Interchange Format, 2013.
- [15] 
- [16] ELLIS, I. *Ellis' British railway engineering encyclopaedia, fourth edition*. Lulu, 2019.
- [17] ERL, T., CARLYLE, B., PAUTASSO, C., AND BALASUBRAMANIAN, R. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. The Prentice Hall service technology series. Pearson Education, 2013.
- [18] EUROPEAN UNION AGENCY FOR RAILWAYS. ERTMS/ETCS Application Level 2. In *ERTMS/ETCS System Requirements Specification*. European Union Agency for Railways, 2016, ch. 2.6.6, pp. 21–22.
- [19] EUROPEAN UNION AGENCY FOR RAILWAYS. ERTMS/ETCS System Requirements Specification, 2016.
- [20] EUROPEAN UNION AGENCY FOR RAILWAYS. Messages. In *ERTMS/ETCS System Requirements Specification*. European Union Agency for Railways, 2016, ch. 8.
- [21] EUROPEAN UNION AGENCY FOR RAILWAYS. Technical Specifications for Interoperability (TSIs). https://www.era.europa.eu/domains/technical-specifications-interoperability_en, 2024. [Online; accessed 2024-09-12].
- [22] FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000, ch. 5.
- [23] GOETZ, B. *Java concurrency in practice*. Pearson Education, 2006.
- [24] GROUP, O. M. Unified modelling language. <https://www.omg.org/spec/UML>, 2017. [Online; accessed 2023-11-23].
- [25] HALL, C., AND HALL, S. *Modern signalling handbook*, 5th edition ed. Ian Allan Publishing Hersham, Hersham, 2016. Section: 1 volume : illustrations ; 21 cm.
- [26] HAXTHAUSEN, A. E., AND ØSTERGAARD, P. H. On the use of static checking in the verification of interlocking systems. In *ISoLA* (2016), LNCS 9953.

-
- [27] HELM, R., JOHNSON, R. E., GAMMA, E., AND VLISSIDES, J. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995, ch. 5, p. 331ff.
- [28] HINMAN, P. G. *Fundamentals of mathematical logic*. CRC Press, 2018.
- [29] HOW, F., ALEXANDER, J., AND BRADBEER, T. Points part 1. In *Back to basics*. Institution of Railway Signal Engineers, 2022.
- [30] IDANI, A., LEDRU, Y., AIT WAKRIME, A., BEN AYED, R., AND BON, P. Towards a tool-based domain specific approach for railway systems modeling and validation. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification* (Cham, 2019), S. Collart-Dutilleul, T. Lecomte, and A. Romanovsky, Eds., Springer International Publishing, pp. 23–40.
- [31] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements. *IEEE Std 1474.1-2004 (Revision of IEEE Std 1474.1-1999)* (2004), 0_1–45.
- [32] INTERNATIONAL UNION OF RAILWAYS. Gsm-r. <https://uic.org/rail-system/telecoms-signalling/article/gsm-r>, 2024. [Online; accessed 2024-04-14].
- [33] JAMES, P., MOLLER, F., AND PANTEKIS, F. Ontrack: Reflecting on domain specific formal methods for railway designs. *Science of Computer Programming* 233 (2024), 103057.
- [34] JAMES, P., AND ROGGENBACH, M. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *Math. Comput. Sci.* 8, 1 (2014), 11–38.
- [35] KITZINGER, J. Audience understandings of aids media messages: a discussion of methods. *Sociology of Health & Illness* 12, 3 (1990), 319–335.
- [36] KITZINGER, J. The methodology of focus groups: the importance of interaction between research participants. *Sociology of Health & Illness* 16, 1 (1994), 103–121.
- [37] KRUEGER, R. A. *Focus groups: A practical guide for applied research*. Sage publications, 2014.
- [38] LADENBERGER, L., BENDISPOSTO, J., AND LEUSCHEL, M. Visualising Event-B Models with B-Motion Studio. In *Formal Methods for Industrial Critical Systems* (Berlin, Heidelberg, 2009), M. Alpuente, B. Cook, and C. Joubert, Eds., Springer Berlin Heidelberg, pp. 202–204.
- [39] LECOMTE, T., BURDY, L., AND LEUSCHEL, M. Formally checking large data sets in the railways, 2012.
- [40] LEUSCHEL, M., AND BUTLER, M. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer* 10, 2 (2008), 185–203.

- [41] LEUSCHEL, M., SAMIA, M., AND BENDISPOSTO, J. Easy graphical animation and formula visualisation for teaching b.
- [42] LUTEBERGET, B. *Automated Reasoning for Planning Railway Infrastructure*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2019.
- [43] MEWES, K. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation*. PhD thesis, University of Bremen, 2010.
- [44] MICROSOFT. The c# type system. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/#the-common-type-system>, 2023. [Online; accessed 2023-11-22].
- [45] MICROSOFT. Data binding overview. <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-6.0>, 2023. [Online; accessed 2023-12-08].
- [46] MICROSOFT. Enumerable.Select Method. <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=net-6.0>, 2023. [Online; accessed 2023-12-01].
- [47] MICROSOFT. An introduction to NuGet. <https://learn.microsoft.com/en-us/nuget/what-is-nuget>, 2023. [Online; accessed 2023-11-23].
- [48] MICROSOFT. IReadOnlyCollection<T> interface. <https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlycollection-1?view=net-6.0>, 2023. [Online; accessed 2023-11-22].
- [49] MICROSOFT. List<T> class. <https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0>, 2023. [Online; accessed 2023-11-22].
- [50] MICROSOFT. Local functions. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions>, 2023. [Online; accessed 2023-11-22].
- [51] MICROSOFT. Methods: Passing by reference vs. passing by value. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods#passing-by-reference-vs-passing-by-value>, 2023. [Online; accessed 2023-11-22].
- [52] MICROSOFT. .NET — introduction and overview. <https://learn.microsoft.com/en-us/dotnet/core/introduction>, 2023. [Online; accessed 2023-12-19].
- [53] MICROSOFT. Organize types in namespaces. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces>, 2023. [Online; accessed 2023-11-30].

-
- [54] MICROSOFT. Overview of ASP.NET Core. <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0>, 2023. [Online; accessed 2023-12-19].
 - [55] MICROSOFT. PackageReference in project files. <https://learn.microsoft.com/en-us/nuget/consume-packages/package-references-in-project-files>, 2023. [Online; accessed 2023-11-23].
 - [56] MICROSOFT. Pattern matching overview. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching>, 2023. [Online; accessed 2023-12-11].
 - [57] MICROSOFT. Projection operations. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations#zip>, 2023. [Online; accessed 2023-11-29].
 - [58] MICROSOFT. Styles and templates. <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/styles-templates-overview?view=netdesktop-6.0>, 2023. [Online; accessed 2023-12-05].
 - [59] MICROSOFT. What's new in C# 10. <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-10>, 2023. [Online; accessed 2023-12-19].
 - [60] MICROSOFT. Windows Presentation Foundation. <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0>, 2023. [Online; accessed 2023-12-05].
 - [61] MICROSOFT. XAML overview. <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-6.0>, 2023. [Online; accessed 2023-12-05].
 - [62] MICROSOFT. XmlSerializer Class. <https://learn.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=net-6.0>, 2023. [Online; accessed 2023-12-19].
 - [63] MICROSOFT RESEARCH. The Z3 Theorem Prover. <https://github.com/Z3Prover/z3>, 2022. [Online; accessed 2022-12-08].
 - [64] MORALES, J. F. jfmc's Z3 Playground. <https://jfmc.github.io/z3-play/>, 2022. [Online; accessed 2022-12-09].
 - [65] .NET FOUNDATION AND CONTRIBUTORS. RestSharp. <https://restsharp.dev/>, 2023. [Online; accessed 2023-12-19].
 - [66] NETWORK RAIL. Symbols for plans and sketches used in signalling applications. Standard, Network Rail, 10 2020.
 - [67] NETWORK RAIL. Command and control system; control system and interlocking platforms: Definitions. Network Rail Standard, 2023.

- [68] NETWORK RAIL. Jargon buster. <https://safety.networkrail.co.uk/tools-resources/jargon-buster/>, 2023. [Online; accessed 2023-12-21].
- [69] NETWORK RAIL. Network statement. <https://www.networkrail.co.uk/industry-and-commercial/information-for-operators/network-statement/>, 2023. [Online; accessed 2023-12-21].
- [70] NETWORK RAIL. Western and wales sectional appendix december 2023. <https://www.networkrail.co.uk/industry-and-commercial/information-for-operators/national-electronic-sectional-appendix/>, 2023. [Online; accessed 2023-12-21].
- [71] NETWORK RAIL. Catalogue of Network Rail Standards. <https://www.networkrail.co.uk/wp-content/uploads/2024/07/Catalogue-of-Network-Rail-Standards-Issue-132.xlsx>, 2024.
- [72] NORRIS, J. *The Bristol & South Wales Union Railway*. Railway and Canal Historical Society Oakham, Leics., Oakham, Leics., 1985. Section: 28 pages : illustrations, 2 maps ; 21 cm.
- [73] PANTEKIS, F., JAMES, P., O'REILLY, L., ARCHAMBAULT, D., AND MOLLER, F. Visualising railway safety verification. In *Formal Techniques for Safety-Critical Systems* (Cham, 2020), O. Hasan and F. Mallet, Eds., Springer International Publishing, pp. 95–105.
- [74] PETROVSKA, O., AND SEISENBERGER, M. Report on BCTCS 2022. *Bull. EATCS* 137 (2022).
- [75] PLATZER, A., AND QUESEL, J.-D. European Train Control System: A Case Study in Formal Verification. In *ICFEM 2009* (2009), LNCS 5885, Springer.
- [76] POWELL, R. A., AND SINGLE, H. M. Focus Groups. *International Journal for Quality in Health Care* 8, 5 (01 1996), 499–504.
- [77] PRC RAIL CONSULTING LTD. Infrastructure | The Railway Technical Website. <http://www.railway-technical.com/infrastructure/>, 2022. [Online; accessed 2022-12-11].
- [78] RAIL SAFETY AND STANDARDS BOARD. Glossary of railway terminology. Standard, Rail Safety and Standards Board, 12 2023.
- [79] ROGGENBACH, M., CERONE, A., SCHLINGLOFF, B.-H., SCHNEIDER, G., AND SHAIKH, S. A. *Formal methods for software engineering—languages, methods, application domains*. Springer, 2020.
- [80] SCALA, G. S., NEMEC, D., NEWTON, E., AND MOH-HASSAN. Command Line Parser Library for CLR and NetStandard. <https://github.com/commandlineparser/commandline>, 2023. [Online; accessed 2023-12-01].
- [81] [REDACTED]
- [82] [REDACTED]

-
- [83] [REDACTED]
- [84] [REDACTED]
- [85] [REDACTED]
- [86] [REDACTED] Fortnightly meetings with [REDACTED] engineers, 2022.
- [87] [REDACTED]
- [88] SMARTBEAR SOFTWARE. OpenAPI Specification. <https://swagger.io/specification/v3/>, 2020. [Online; accessed 2023-11-30].
- [89] SMARTBEAR SOFTWARE. Swagger Codegen. <https://swagger.io/tools/swagger-codegen/>, 2023. [Online; accessed 2023-11-30].
- [90] STANSIFER, R. Presburger's article on integer arithmetic: Remarks and translation. Tech. rep., Cornell University, 1984.
- [91] UK RAIL RESEARCH AND INNOVATION NETWORK. UKRRIN. <https://www.ukrrin.org.uk/>, 2023. [Online; accessed 2023-03-14].
- [92] UNIFE. Factsheet #14 - ERTMS Deployment in the UK. <https://www.ertms.net/wp-content/uploads/2018/10/14-ERTMS-deployment-in-the-UK.pdf>, 2018. [Online; accessed 5 Oct 2020].
- [93] UNIFE. ERTMS in brief. https://www.ertms.net/?page_id=40, Oct 2020. [Online; accessed 29 Oct 2020].
- [94] VAN DEN BRAND, M., VAN DEURSEN, A., HEERING, J., DE JONG, H., DE JONGE, M., KUIPERS, T., KLINT, P., MOONEN, L., OLIVIER, P. A., SCHEERDER, J., VINJU, J. J., VISSER, E., AND VISSER, J. The asf+sdf meta-environment: a component-based language development environment. In *First Workshop on Language Descriptions, Tools and Applications, LDTA 2001, a Satellite Event of ETAPS 2001, Genova, Italy, April 7, 2001* (2001), M. van den Brand and D. Parigot, Eds., vol. 44 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 3–8.
- [95] VON SCHOMBERG, R. *A Vision of Responsible Research and Innovation*. John Wiley & Sons, Ltd, 2013, ch. 3, pp. 51–74.
- [96] WOMEN IN RAIL. Industry survey report. Tech. rep., WR, 12 2015.

Appendix A

Glossary of Railway Terms

In this glossary, we explain the railway terminology used in the thesis. These definitions are in the context of the GB main line network owned, maintained and operated by Network Rail. Most of the definitions are taken from [67], [68] and [16].

A.1 General Railway Terms

A.1.1 Infrastructure

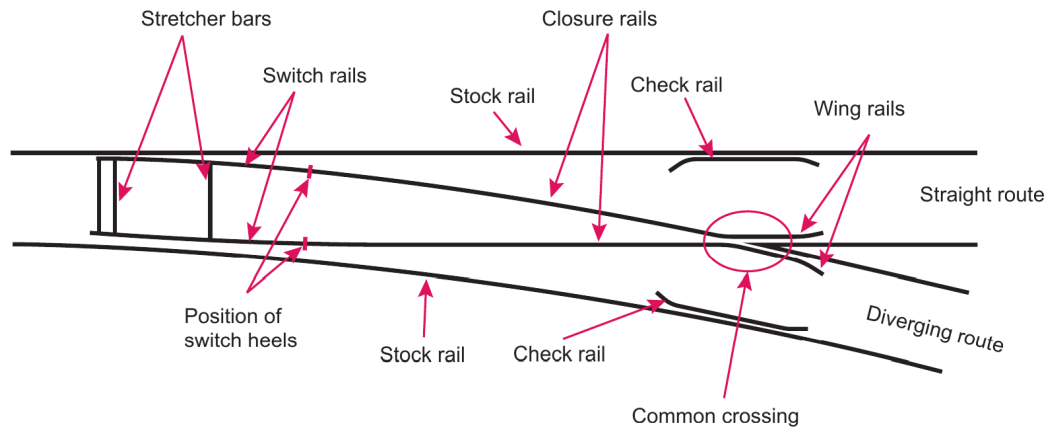


Figure A.1: The principal part of a set of points, from [29].

Engineer's Line Reference (ELR) A three-letter code (sometimes suffixed by a number) that uniquely identifies a railway line in Great Britain. The first ELR — **MLN** — was assigned to the Great Western Railway's *Main Line* between London Paddington and Penzance [12].

Datum A location designated as the "starting point" of a railway line — for example, the datum of the **MLN** line is London Paddington station.

If we think of a railway line as a river and its datum as the source of that river, we can derive the most common naming of **directions** on the GB main line network:

Up The direction towards a datum.

Down The direction leading away from a datum.

Points A movable piece of track that can guide a train onto one of two possible routes, or allow two lines to merge into one. The main parts of a set of points is shown in Figure A.1 [29].

Frog Refers to the *common crossing* in a set of points.

Toes Refers to the end of *switch rails* (where they meet the *stock rails*).

(Diamond) Crossing A fixed at-grade crossing between two tracks, which does not allow trains to travel from one track to the other.

A.1.2 Signalling

Scheme Plan A graphical representation of a section of railway from a signalling perspective. Shows the track layout, positioning of signals and related equipment (e.g. train protection systems).

Route The reserved wheeled path between a defined entry point and a defined exit point, for which *movement authorities* (see below) can be issued.

Control Tables A tabular list of the controlled functions — for example signals, points and level crossings — and how they interact to operate the signalling system. In other words, a control table describes the exact preconditions for setting a route.

Movement Authority (MA) Permission for a train to enter a section of railway, as far as a designated location (e.g. a signal, marker board, or buffer stop). In a conventional signalling system, MAs are communicated to the train driver via lineside signals; In an in-cab signalling system, MAs are displayed on the driver's console onboard the train.

Interlocking Signalling equipment that applies the logic described in control tables. When the signaller requests a route to be set for a train, the interlocking checks that all conditions described in the route's control table are met, changes the state of the infrastructure as needed (e.g. point positions), and applies required locking between functions, before issuing an MA to the train.

Interlockings may be implemented as mechanical frames, relay circuits, electronic circuits, or specialized software.

European Rail Traffic Management System (ERTMS) A signalling and train control system specified by the European Union Agency for Railways [19], intended to improve interoperability between national railway systems in Europe. The system is composed of the **European Train Control System (ETCS)**, its signalling component, as well as

the *Global System for Mobile Communications — Railway (GSM-R)* for its radio communications [32].

Communication-Based Train Control (CBTC) A category of automatic train control systems characterized by [31]:

- Continuous reporting of train location using bidirectional train-to-wayside telecommunications.
- Train-borne and wayside processors capable of implementing automatic train protection (ATP) functions.
- Optional automatic train operation (ATO) and automatic train supervision (ATS) functions.

A.2 Terms Specific to [REDACTED]

In addition to the general railway terms above, [REDACTED] additionally defines the following terms in the context of its Track-Distance Model [81].

Track Representation of a pair of rails that form a continuous line of travel for a train. Note that a track can continue across points and crossings.

Position A named location on the railway, with track, distance and (optionally) direction attributes.

Path Representation of a directed valid journey that a train can make over portions of one or more tracks from an entrance position to an exit position.

Area An extent of the railway made up of one or more portions of tracks. An area does not have to be contiguous, i.e. the tracks involved don't have to be connected.

Appendix B

Files Demonstrating Data Formats

This appendix has been removed for intellectual property reasons.

Appendix C

Source Code of the Z3 Toy Example

Listing C.1: The original scheme plan, which violates the property and generates an unsat core.

```
1 (set-option :produce-unsat-cores true)
2
3 (declare-const b1_offset Int)
4 (declare-const b2_offset Int)
5 (declare-const b3_offset Int)
6
7 (assert (! (= b1_offset 30) :named def_b1 ))
8 (assert (! (= b2_offset 140) :named def_b2 ))
9 (assert (! (= b3_offset 100) :named def_b3 ))
10
11 (push) ; Add a level to the assertion stack
12 (echo "Checking b1 and b2")
13 (assert (!
14   (>= (abs (- b1_offset b2_offset)) 50)
15   :named subprop
16 )) ; Put subprop on the top level
17 (check-sat) ; Check satisfiability of formulae on the stack
18 (pop) ; Remove the top level
19
20 (push)
21 (echo "Checking b1 and b3")
22 (assert (!
23   (>= (abs (- b1_offset b3_offset)) 50)
24   :named subprop
25 ))
26 (check-sat)
27 (pop)
```

```
28
29 (push)
30 (echo "Checking b2 and b3")
31 (assert (!
32   (>= (abs (- b2_offset b3_offset)) 50)
33   :named subprop
34 ))
35 (check-sat)
36 (get-unsat-core)
37 (pop)
```

Listing C.2: The scheme plan with the position of balise b_2 corrected, so that all three sub-properties would produce a sat output.

```
1 (set-option :produce-unsat-cores true)
2
3 (declare-const b1_offset Int)
4 (declare-const b2_offset Int)
5 (declare-const b3_offset Int)
6
7 (assert (! (= b1_offset 30) :named def_b1 ))
8 (assert (! (= b2_offset 150) :named def_b2 ))
9 (assert (! (= b3_offset 100) :named def_b3 ))
10
11 (push)
12 (echo "Checking b1 and b2")
13 (assert (!
14   (>= (abs (- b1_offset b2_offset)) 50)
15   :named subprop
16 ))
17 (check-sat)
18 (pop)
19
20 (push)
21 (echo "Checking b1 and b3")
22 (assert (!
23   (>= (abs (- b1_offset b3_offset)) 50)
24   :named subprop
25 ))
26 (check-sat)
27 (pop)
28
29 (push)
30 (echo "Checking b2 and b3")
```

```

31 (assert (!
32   (>= (abs (- b2_offset b3_offset)) 50)
33   :named subprop
34 ))
35 (check-sat)
36 (pop)

```

Listing C.3: The corrected scheme plan, with the subproperties negated to verify that it complies with the property.

```

1 (set-option :produce-unsat-cores true)
2
3 (declare-const b1_offset Int)
4 (declare-const b2_offset Int)
5 (declare-const b3_offset Int)
6
7 (assert (! (= b1_offset 30) :named def_b1 ))
8 (assert (! (= b2_offset 150) :named def_b2 ))
9 (assert (! (= b3_offset 100) :named def_b3 ))
10
11 (push)
12 (echo "Checking b1 and b2")
13 (assert (!
14   (not
15     (>= (abs (- b1_offset b2_offset)) 50)
16   )
17   :named negated_subprop
18 ))
19 (check-sat)
20 (pop)
21
22 (push)
23 (echo "Checking b1 and b3")
24 (assert (!
25   (not
26     (>= (abs (- b1_offset b3_offset)) 50)
27   )
28   :named negated_subprop
29 ))
30 (check-sat)
31 (pop)
32
33 (push)
34 (echo "Checking b2 and b3")

```

C. Source Code of the Z3 Toy Example

```
35 (assert (!
36   (not
37     (>= (abs (- b2_offset b3_offset)) 50)
38   )
39   :named negated_subprop
40 ))
41 (check-sat)
42 (pop)
```

Appendix D

Focus Group Material

D.1 Briefing Documents

D.1.1 Participant Information Sheet

PARTICIPANT INFORMATION SHEET

(Version 2.0, Date: 27 May 2022)

Project Title:

Visualising Counterexamples for Railway Constraint Checking
Data Error Visualization Focus Group

Contact Details:

Victor Cai, Department of Computer Science, Swansea University
[redacted]

Supervisors:

Markus Roggenbach
[redacted]

Monika Seisenberger
[redacted]

1. Invitation Paragraph

Welcome to the focus group on the graphical representation of design rule violations. It is a part of my project on extending the [redacted] editor and the [redacted] [redacted] to display these violations. As people who directly work on designing track plans and the editor, your input is much appreciated.

2. What is the purpose of the study?

We hope to find an intuitive graphical representation of design rule violations, for each category of design rules that concern balise groups (BGs).

3. Why have I been chosen?

You were chosen because your work involves designing track plans, and so you are the stakeholders of the editor and the data [redacted]

Participation in this focus group is voluntary, [redacted] you are free to withdraw at any time if you wish.

4. What will happen to me if I take part?

- 1) You will be split into three groups.
- 2) You will each be presented with a number of track plans, each with a design rule violation, and a textual description of the violation. Your task is to:
 - a. Annotate each track plan in the way you think the editor should highlight the rule violation.
 - b. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 - c. Describe how you want the editor to display the textual description.
- 3) Each group agrees on one visualisation for each of the track plans.

The three groups then come together for a plenary discussion. For each violation:

- 4) Each group presents their chosen presentation from step 3.
- 5) We discuss the pros & cons of each proposed visualisation
- 6) If possible, we agree upon a common presentation.

Your responses in the plenary discussion will be recorded and analysed, and a summary of the responses we receive will be kept for reference in later stages of this project.

5. What are the possible disadvantages of taking part?

The plenary session will be recorded on video. We will only do this with your explicit permission (by signing the consent form). The video will only be used internally for analysing the outcomes of this focus group study, and will not be published.

6. What are the possible benefits of taking part?

Your feedback will help us build an intuitive user interface to display violations of design rules to railway engineers. By making the representation easily understandable by the engineers, we hope the software tool will help with the infrastructure design process in the future.

7. Will my taking part in the study be kept confidential?

Your responses will be recorded in an anonymised form. Information collected in this focus group study will only be available to Victor, Markus, Monika, [redacted] [redacted] and Dave. Our summarised findings may appear in Victor's dissertation.

8. What if I have any questions?

Please contact us using the email addresses at the beginning of this information sheet.

D.1.2 Consent Form

PARTICIPANT CONSENT FORM
(Version 1.0, Date: 27/05/2022)

Project Title:

Data Error Visualization Focus Group

Contact Details:

Victor Cai, Department of Computer Science, Swansea University
[REDACTED]

Please initial box

- | | | |
|---|--------------------------|--|
| 1. I confirm that I have read and understood the information sheet dated 27 May 2022 (version number 2.0) for the above study and have had the opportunity to ask questions. | ☐ | |
| 2. I understand that my participation is voluntary and that I am free to withdraw at any time, without giving any reason, without my medical care or legal rights being affected. | ☐ | |
| 3. I understand that sections of any of data obtained may be looked at by responsible individuals from Swansea University or from regulatory authorities where it is relevant to my taking part in research. I give permission for these individuals to have access to these records. | ☐ | |
| 4. I agree to be included in the video recording for the plenary discussion. The recording will only be used for the focus group evaluation, and will be deleted afterwards. | ☐ | |
| 5. I agree to take part in the above study. | | |

Name of Participant	Date	Signature
Researcher	Date	Signature

Personal data collected on this form will be processed in line with the General Data Protection Regulation 2016 and the Data Protection Act 2018. Further information about how your data is managed is available on the [University Research Privacy Notice](#).

<https://www.swansea.ac.uk/about-us/compliance/data-protection/research-privacy-notice/>

D.2 Briefing Presentation

180

Data Error Visualisation Focus Group

Victor Cai¹, [REDACTED]
Markus Roggenbach¹, Monika Seisenberger¹

Swansea University

[REDACTED] Ltd.

6 June 2022

What happens today: group exercise

- ▶ You will be split into three groups

What happens today: group exercise

- ▶ You will be split into three groups
- ▶ Within your group, you will work on 5 track plans . . .

Author: _____	Version: _____	Task 1
---------------	----------------	--------

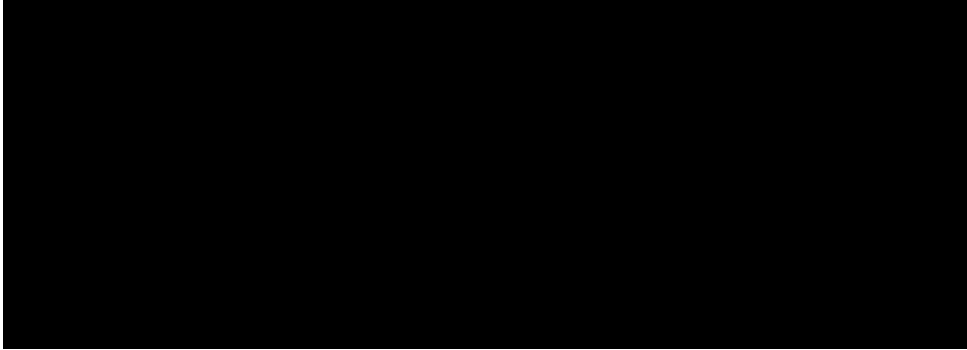
Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.
Think about:
 - How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
3. Describe how you want the editor to display the textual description.

Your preferred textual description: _____
How the editor should display the description: _____
Any other suggestions: _____

What happens today: group exercise

- ▶ You will be split into three groups
- ▶ Within your group, you will work on 5 track plans . . .

Author: _____	Version: _____	Task 1
Instructions		
1. Annotate each track plan in the way you think the editor should highlight the rule violation. Think about:		
• How to highlight the elements involved. • How to represent the specific nature of the rule.		
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).		
3. Describe how you want the editor to display the textual description.		
		
Your preferred textual description: _____		
How the editor should display the description: _____		
Any other suggestions: _____		

- ▶ Each group agrees on one visualisation for each of the track plans.

What happens today: plenary discussion

For each violation:

- ▶ Each group presents their chosen presentation

What happens today: plenary discussion

For each violation:

- ▶ Each group presents their chosen presentation
- ▶ We discuss the pros & cons of each proposed visualisation

What happens today: plenary discussion

For each violation:

- ▶ Each group presents their chosen presentation
- ▶ We discuss the pros & cons of each proposed visualisation
- ▶ If possible, we agree upon a common presentation

Important: video recording

The plenary session will be recorded on video. **We will only do this with your explicit permission (by signing the cosent form).** The video will only be used internally for analysing the outcomes of this focus group study, and will not be published.

What happens after this focus group

- ▶ **By August:** Swansea team will analyse the outcomes
- ▶ **By October:** Implement visualisations for the editor
- ▶ **November / December:** You may be invited for an acceptance test

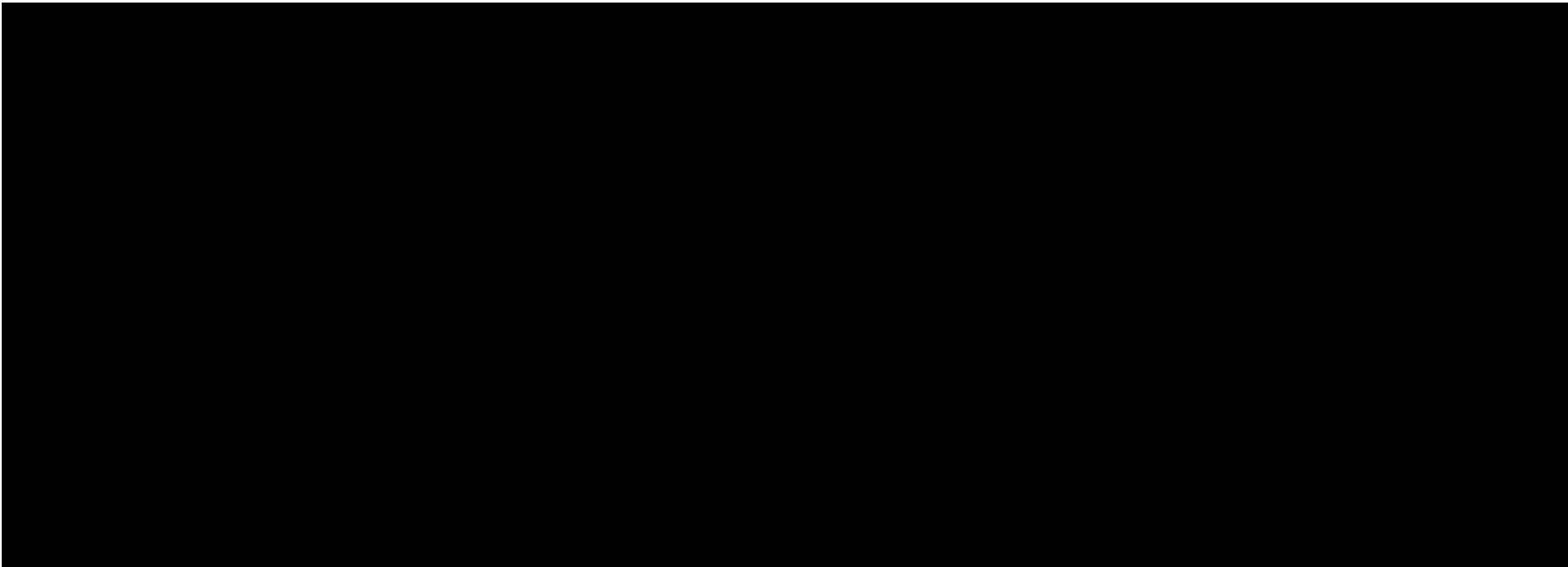
D.3 Exercise Sheets

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

Instructions

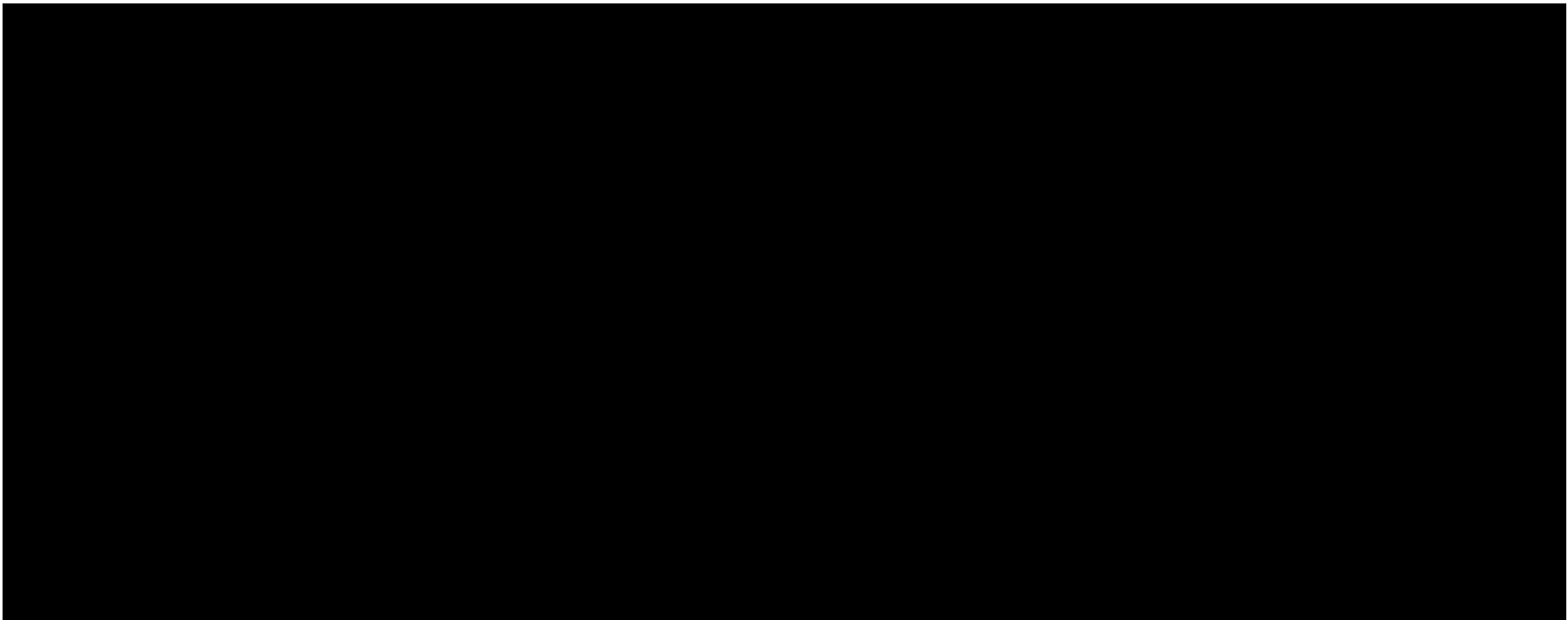
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
- How to represent the specific nature of the rule.

2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).

3. Describe how you want the editor to display the textual description.



Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

Instructions

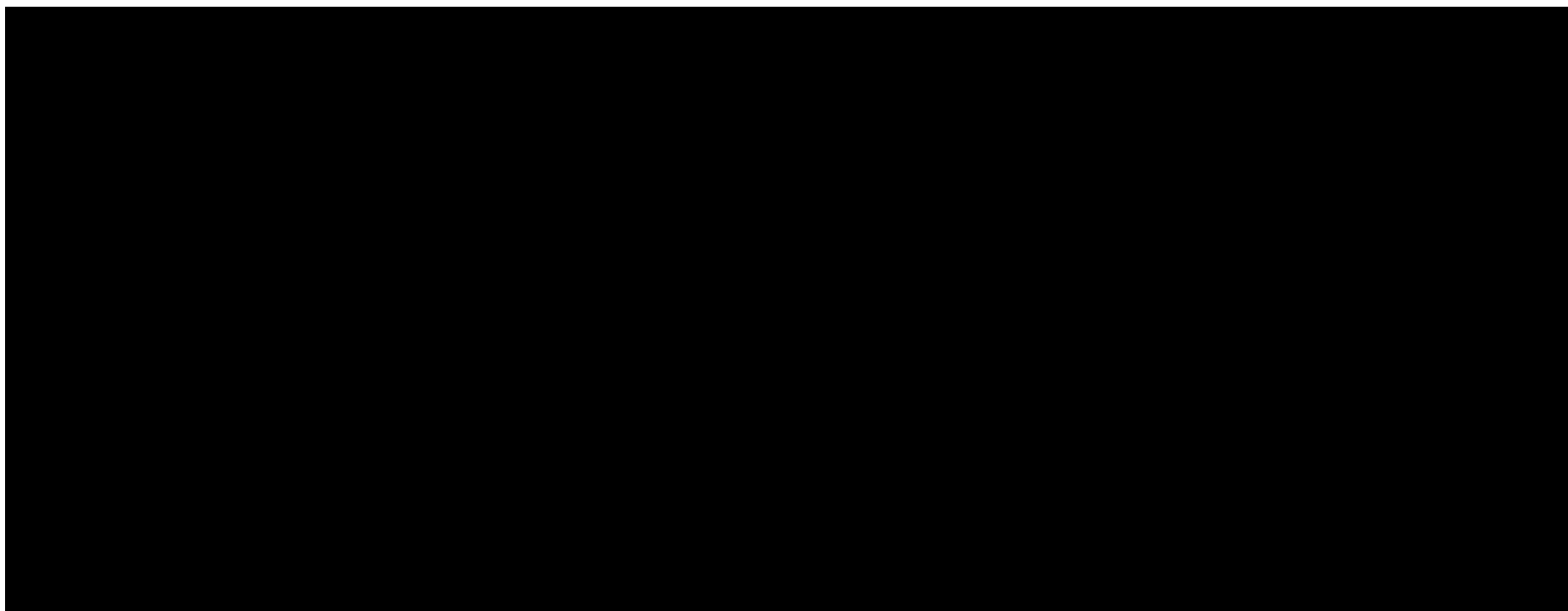
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
- How to represent the specific nature of the rule.

2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).

3. Describe how you want the editor to display the textual description.



Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

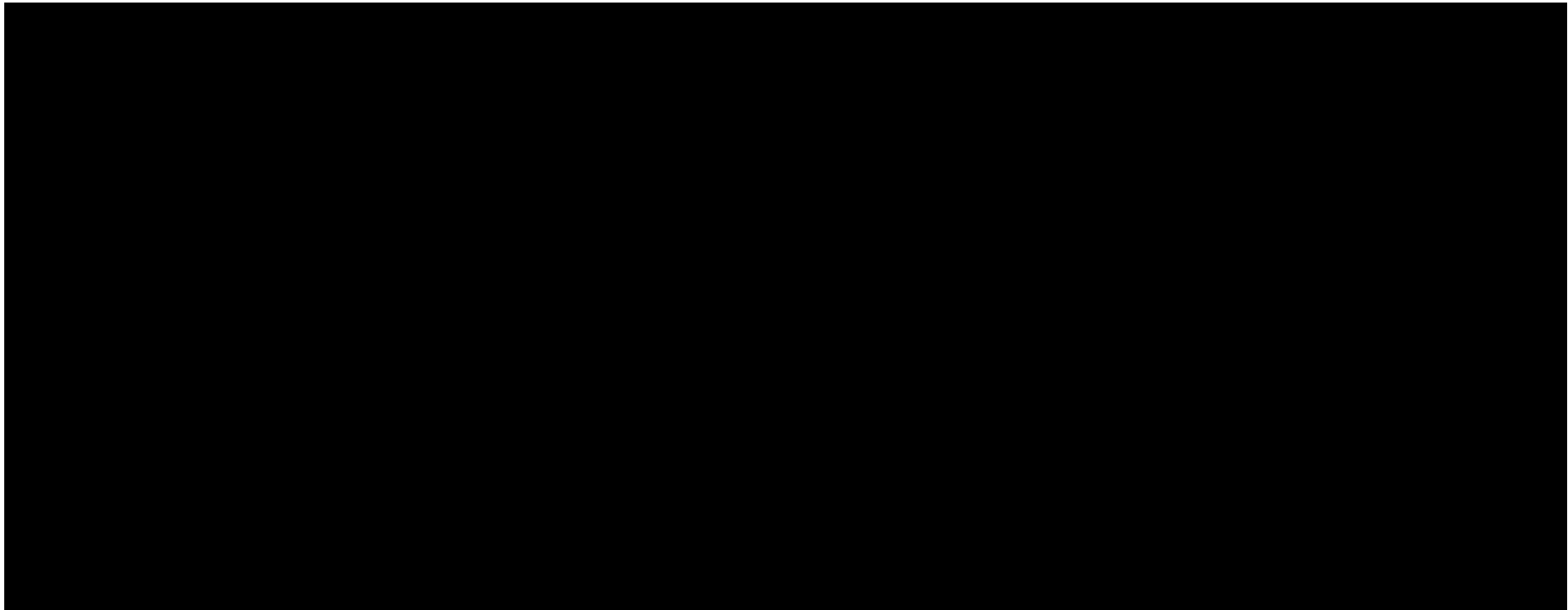
Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
- How to represent the specific nature of the rule.

2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
3. Describe how you want the editor to display the textual description.



Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

Instructions

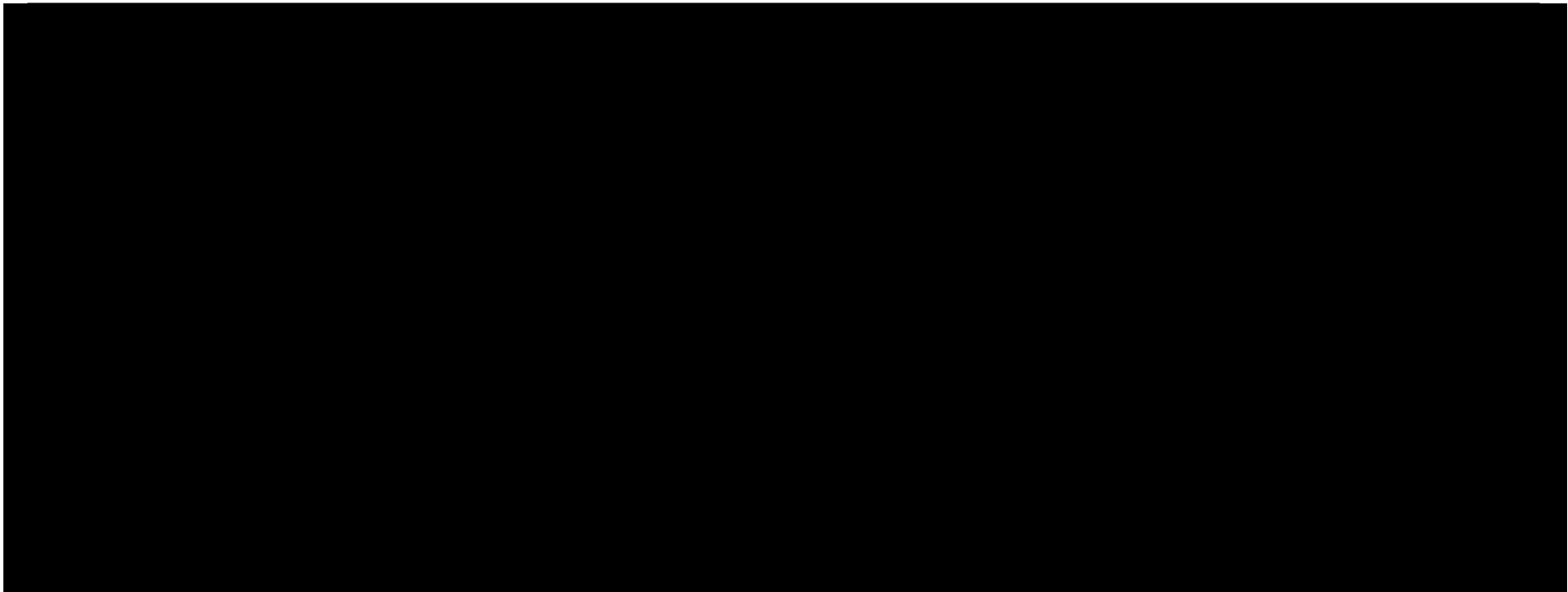
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
- How to represent the specific nature of the rule.

2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).

3. Describe how you want the editor to display the textual description.



Your preferred textual description: _____

How the editor should display the description: _____

Any other suggestions: _____

D.4 Completed Exercise Sheets

Group 1

Author: ①

Version: _____

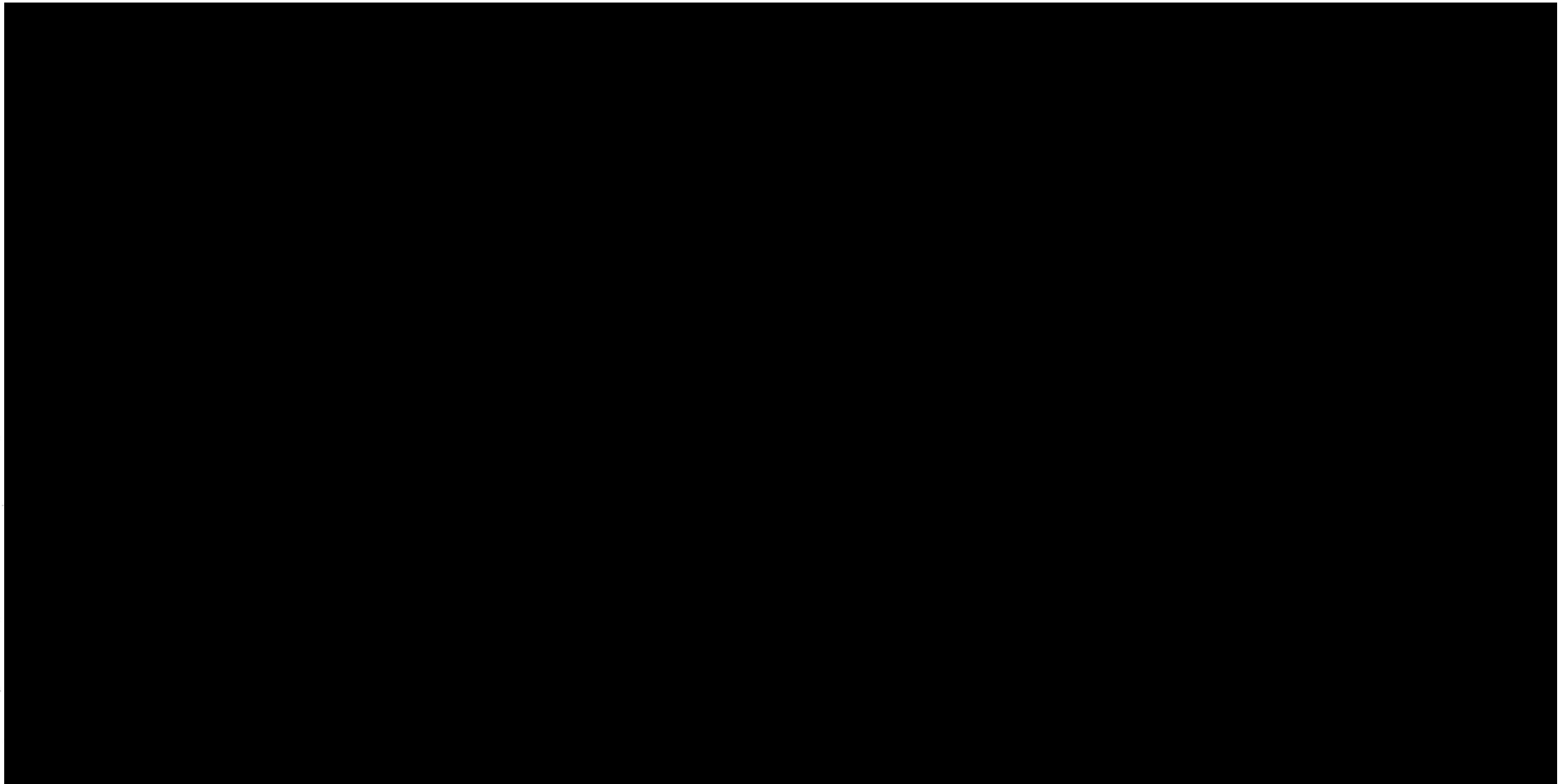
Task 1

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

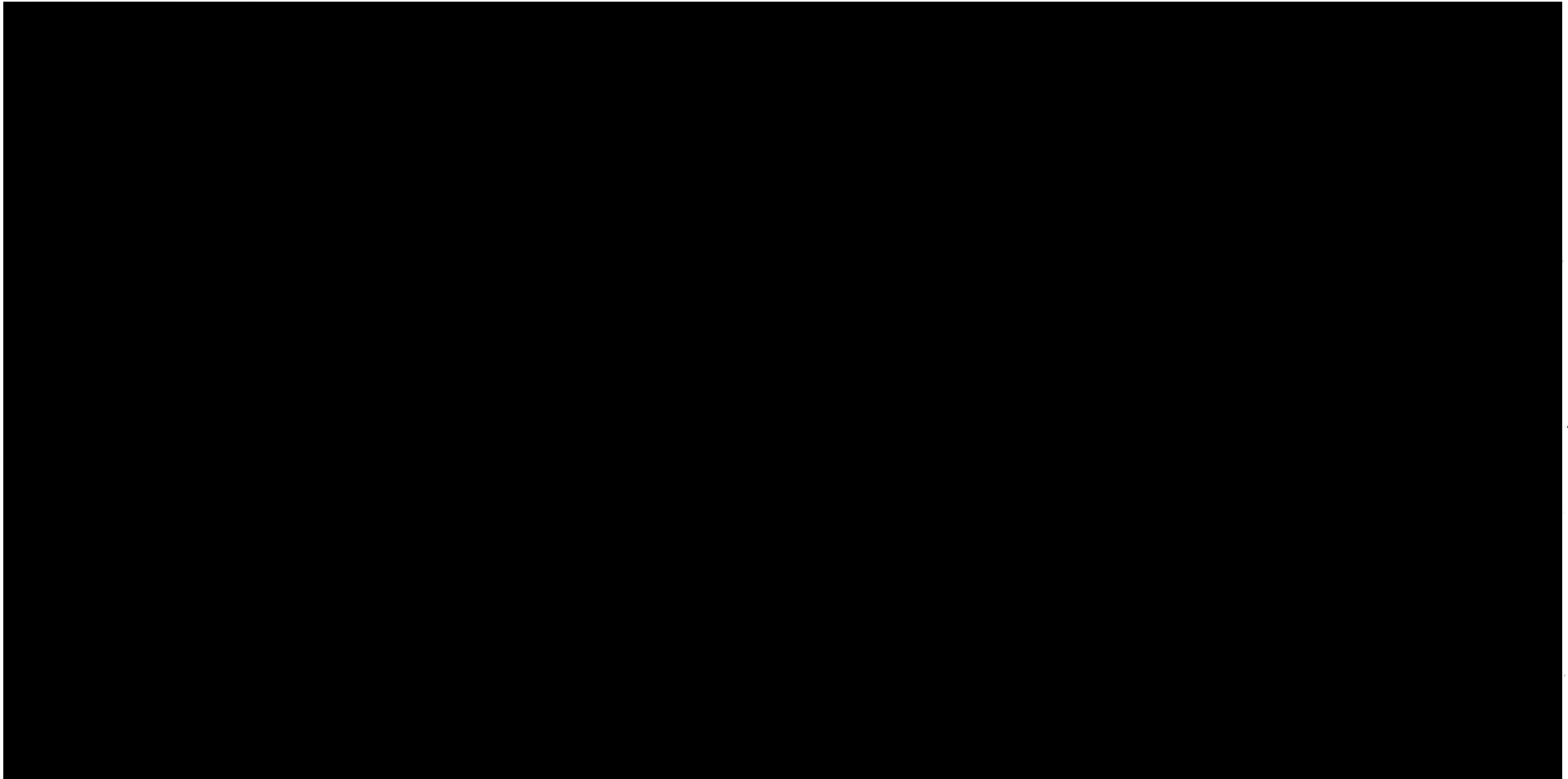


Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



Author: _____

1

Version: _____

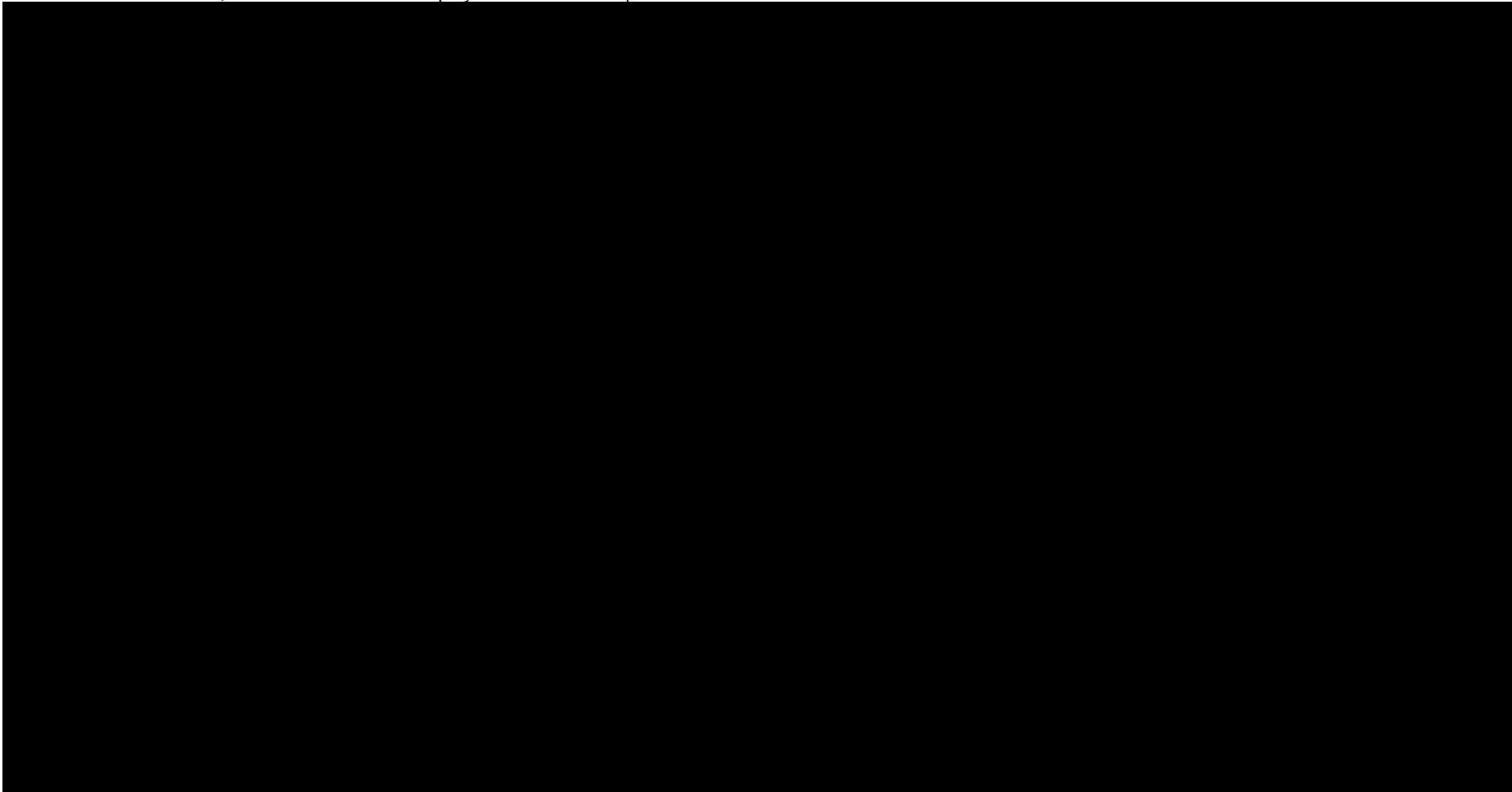
Task 3

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



Author: _____ Version: _____

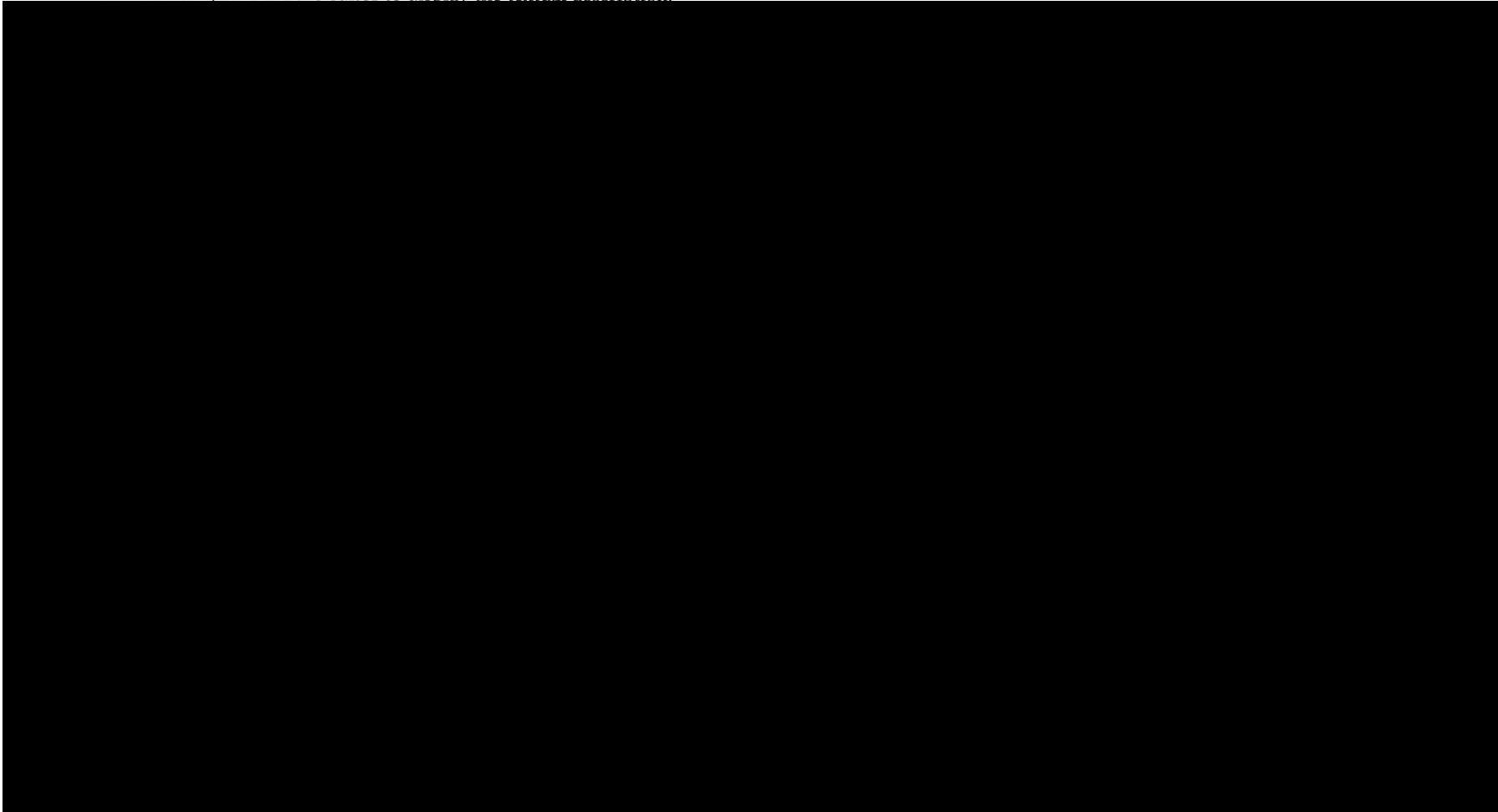
Task 4

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



Author: _____

①

Version: _____

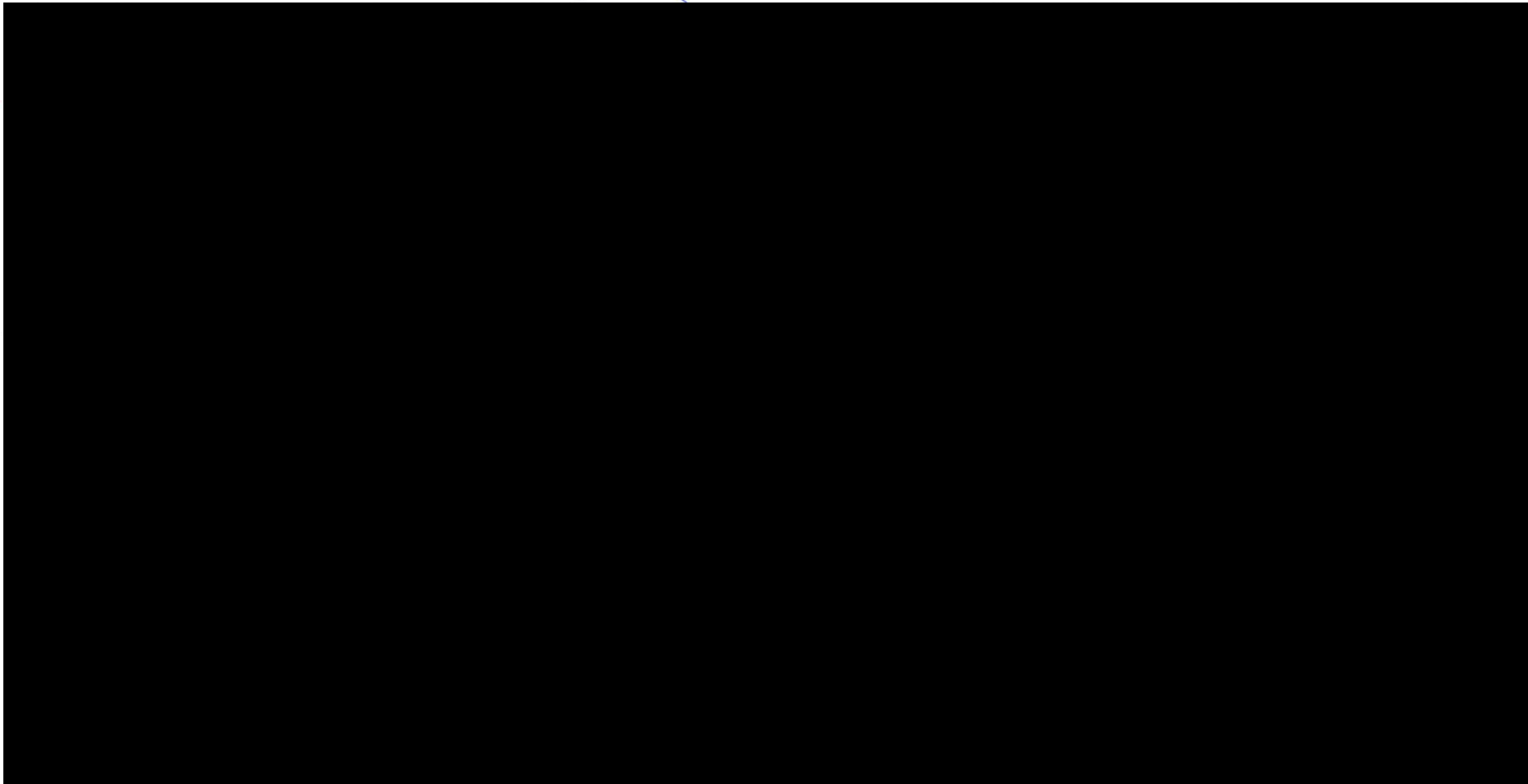
Task 5

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



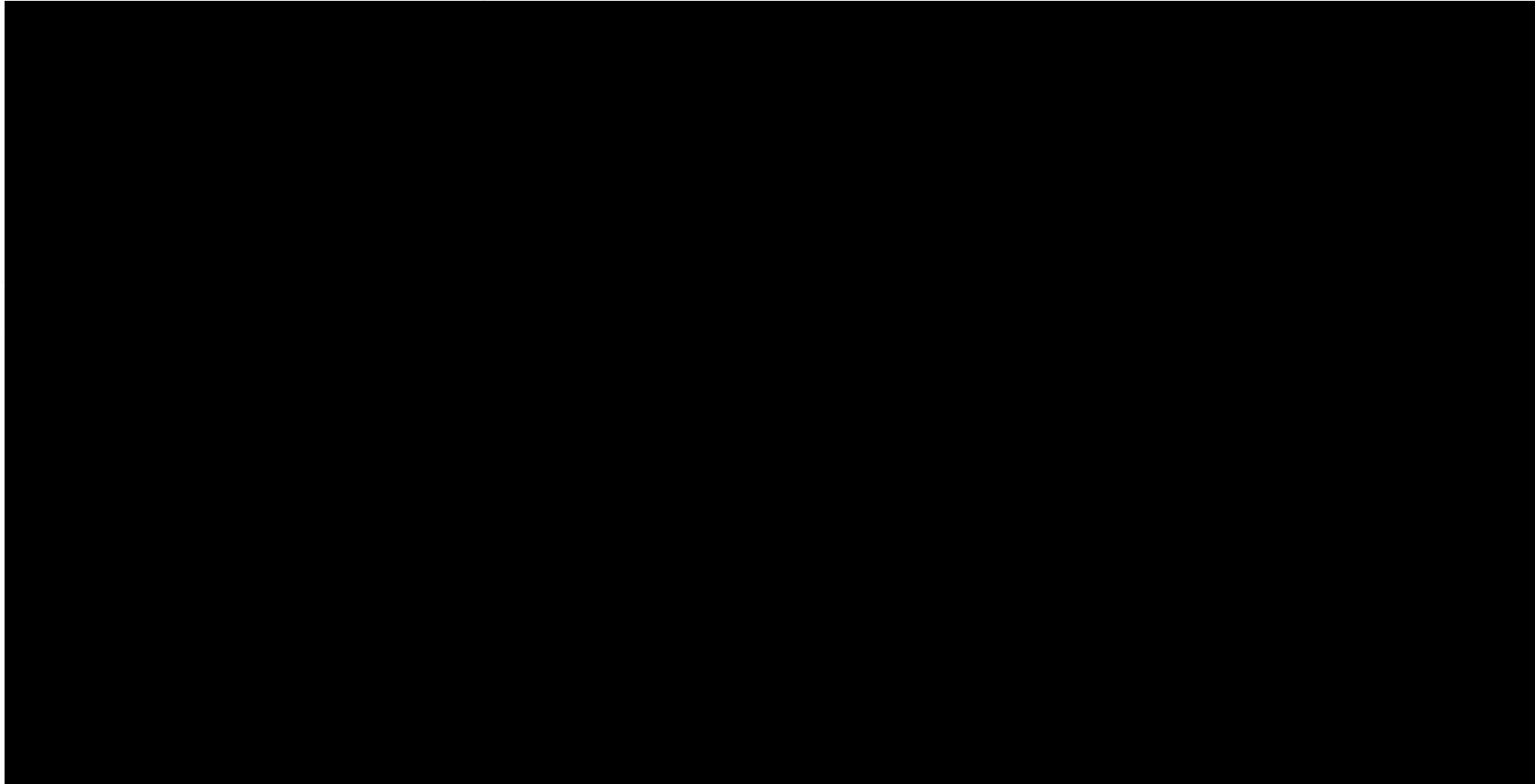
Group 2

Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

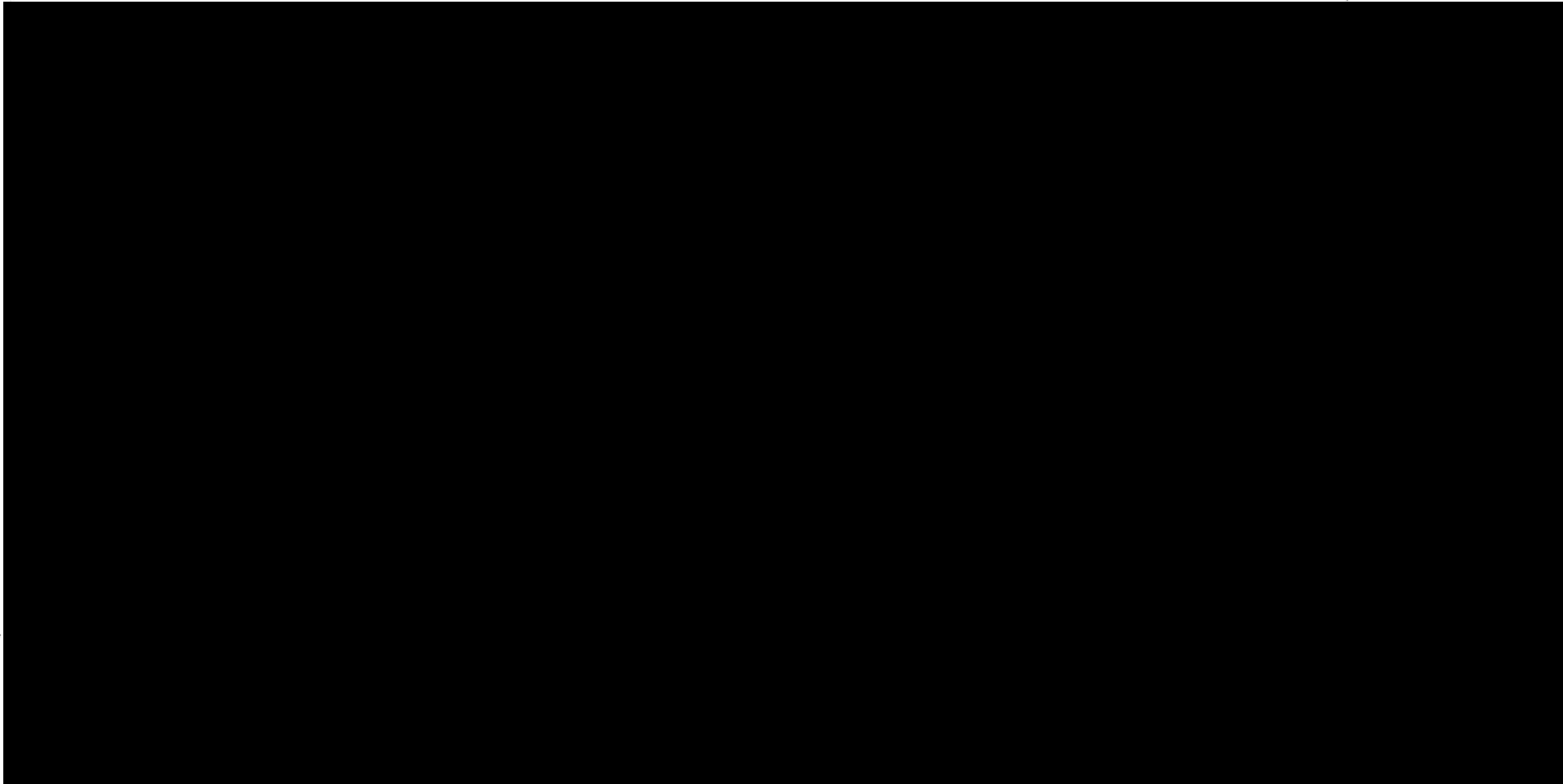


Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

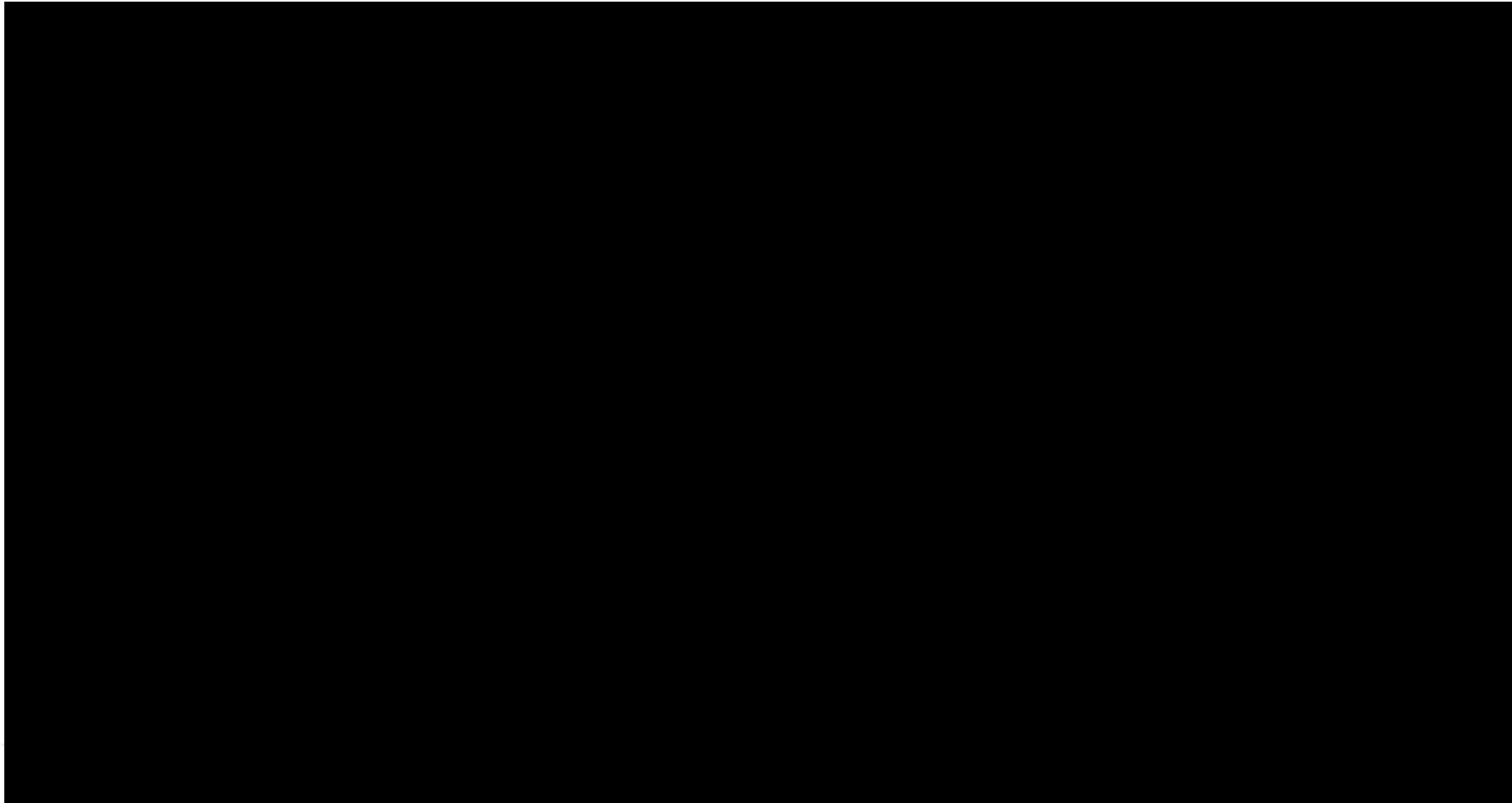


Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

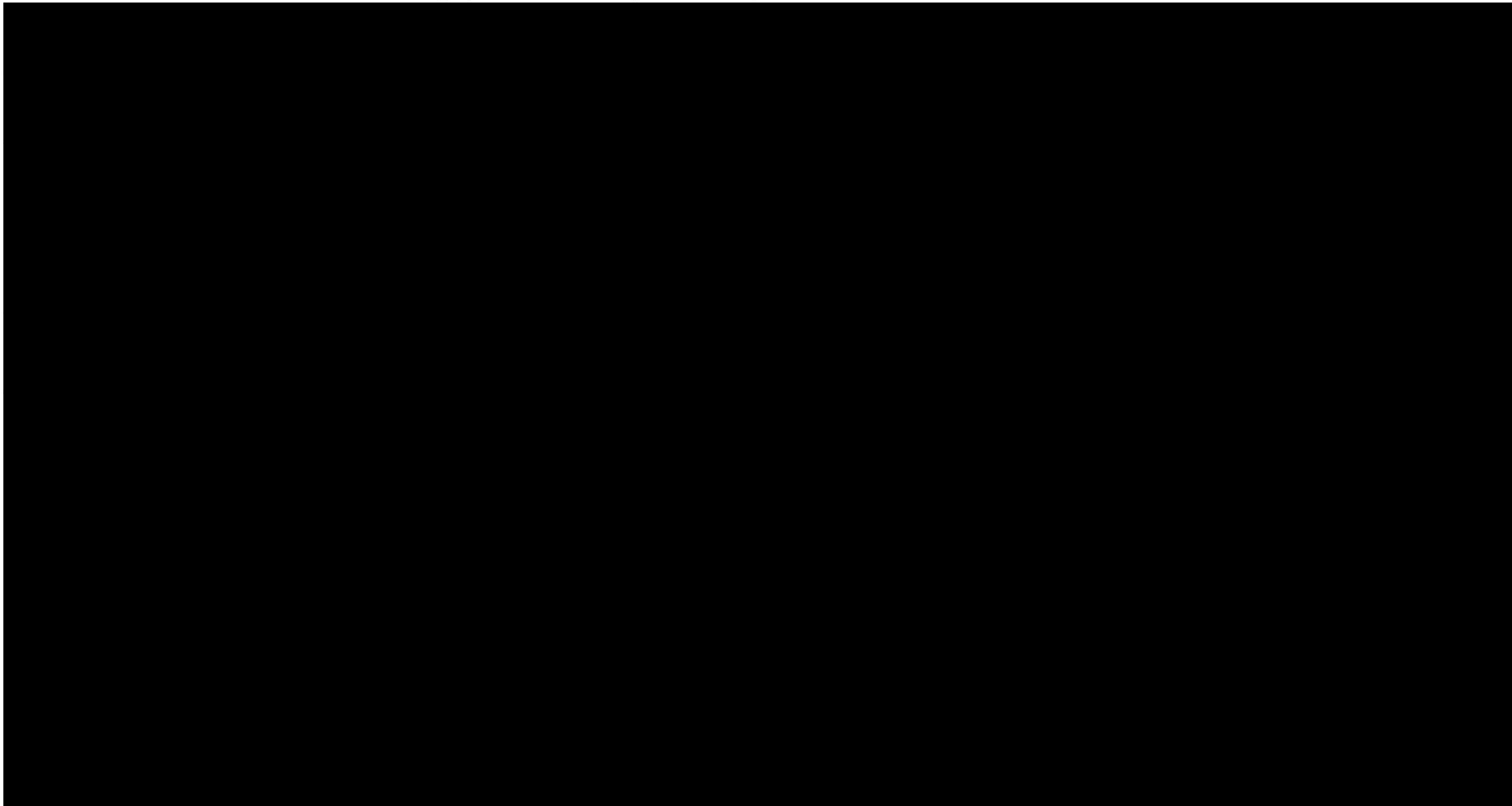


Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

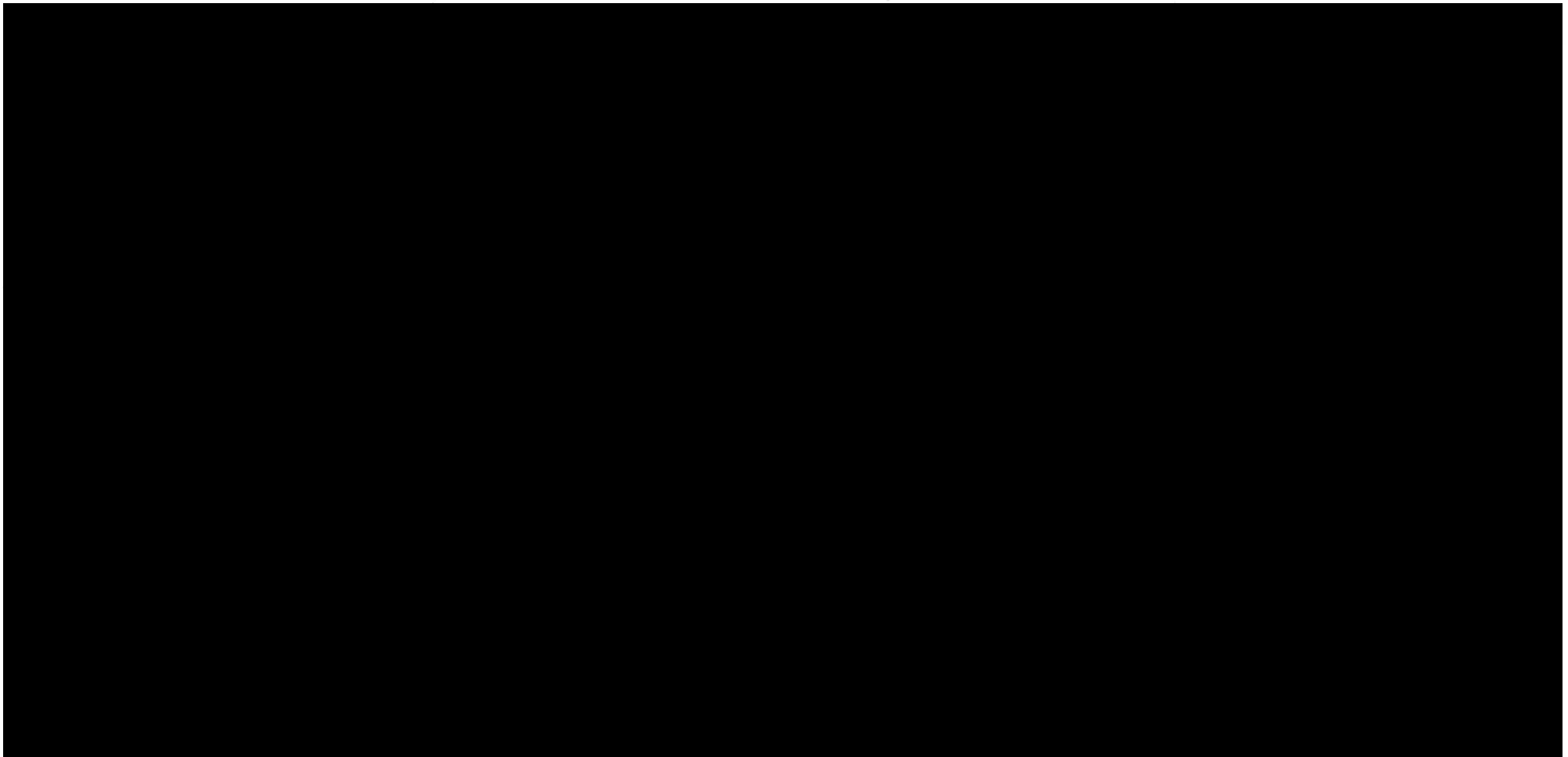


Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.



Group 3

Author: _____

63

Version: _____

Solution

Task 1

Instructions

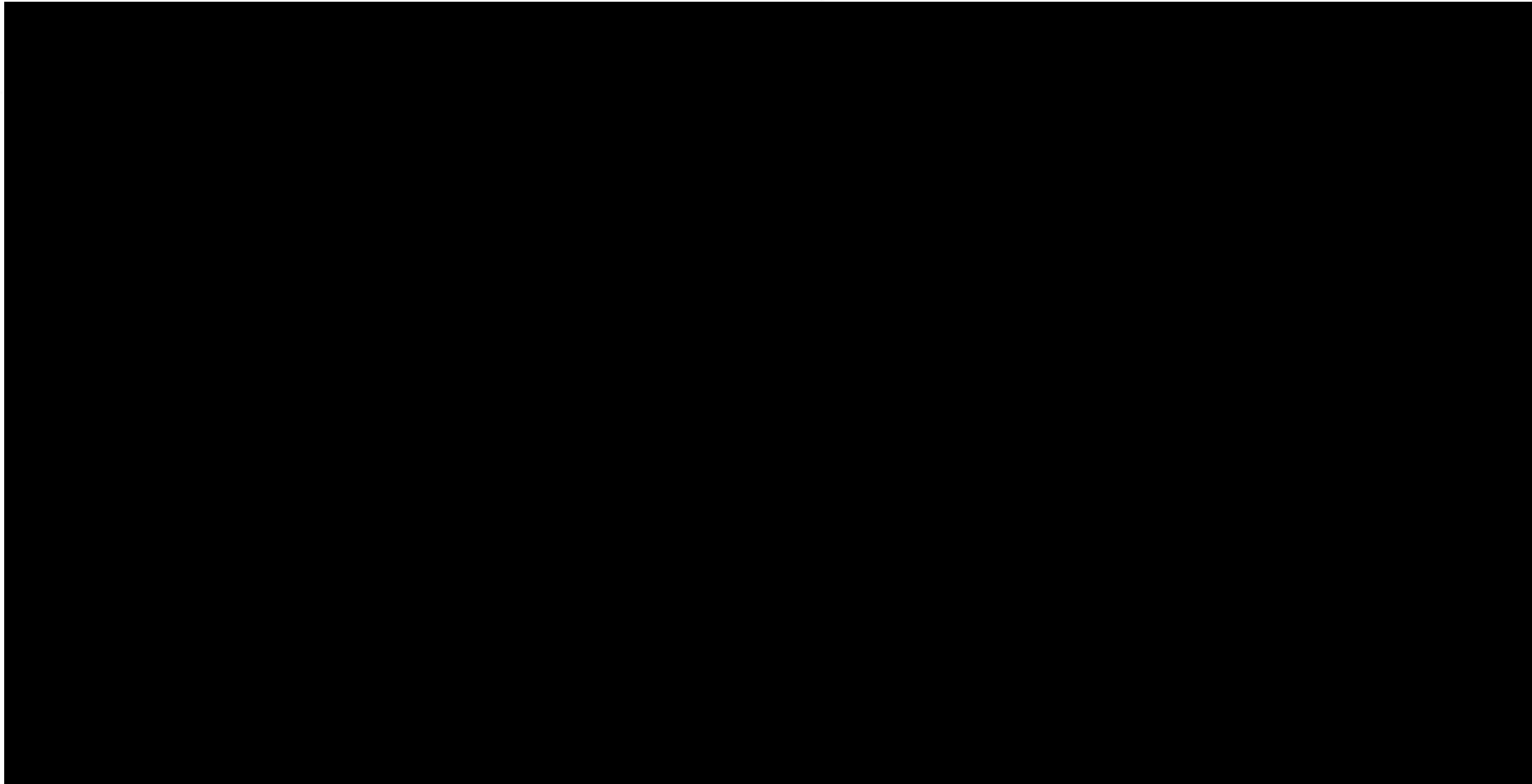
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

Step 1

Area Indication



Author: _____

G3

Version: _____

Solution

Task 1

Instructions

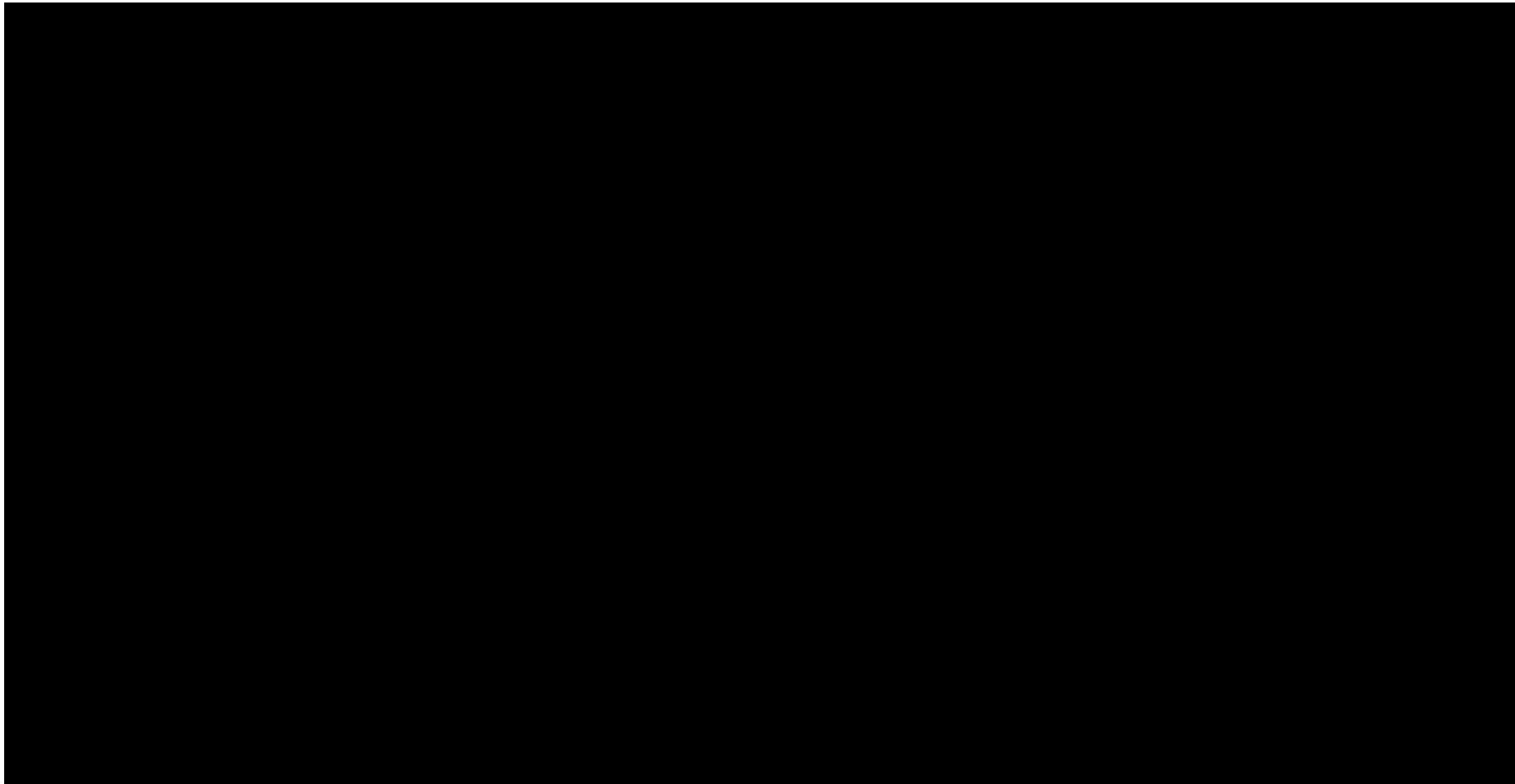
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

Step 2

Flash or highlight
the elements in
error



Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

Step 3

- ✗ List the errors for the element
- ✗ Remove irrelevant objects from display

Instructions

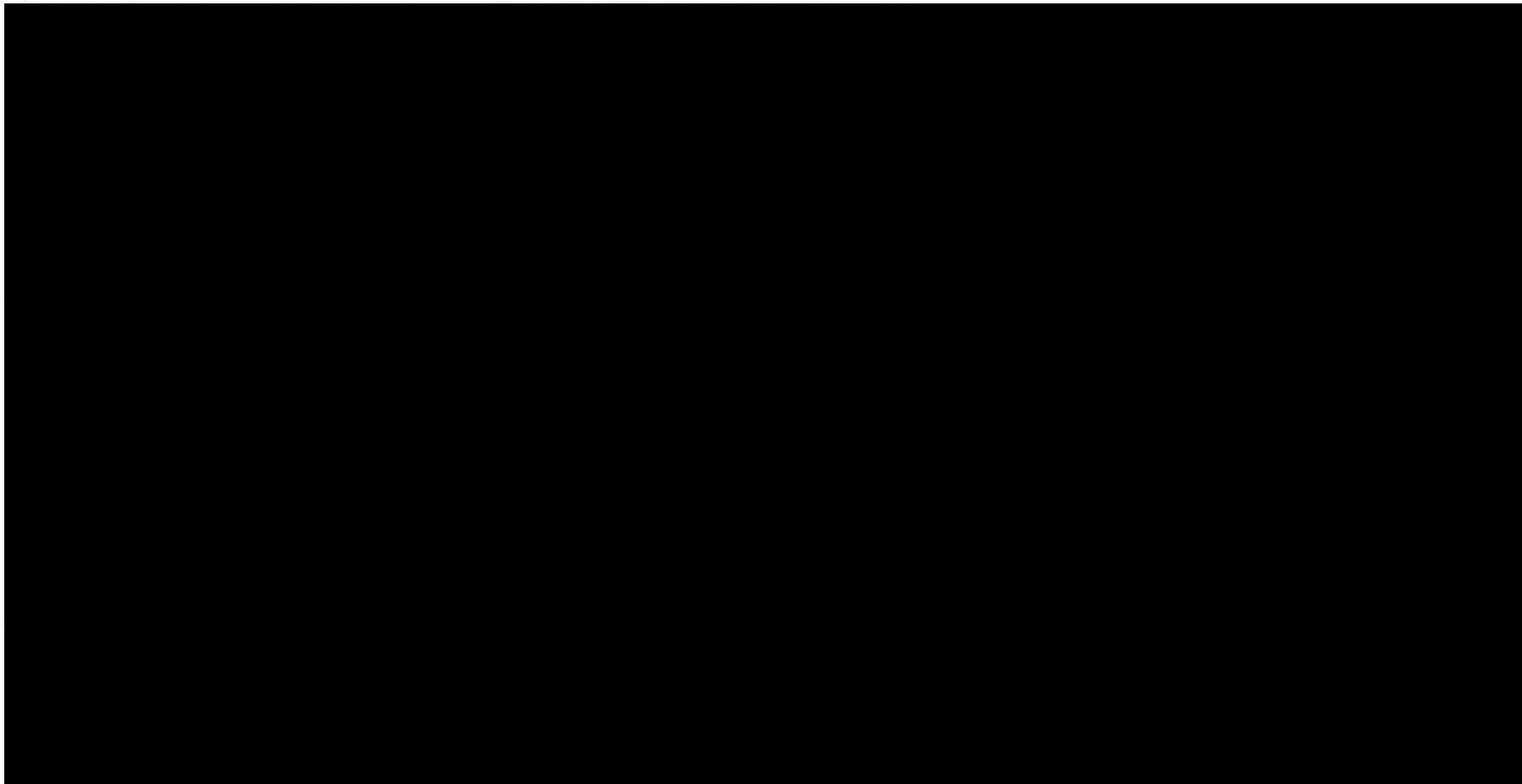
1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
- How to represent the specific nature of the rule.

2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
3. Describe how you want the editor to display the textual description.

Step 4
* highlight the path
in error



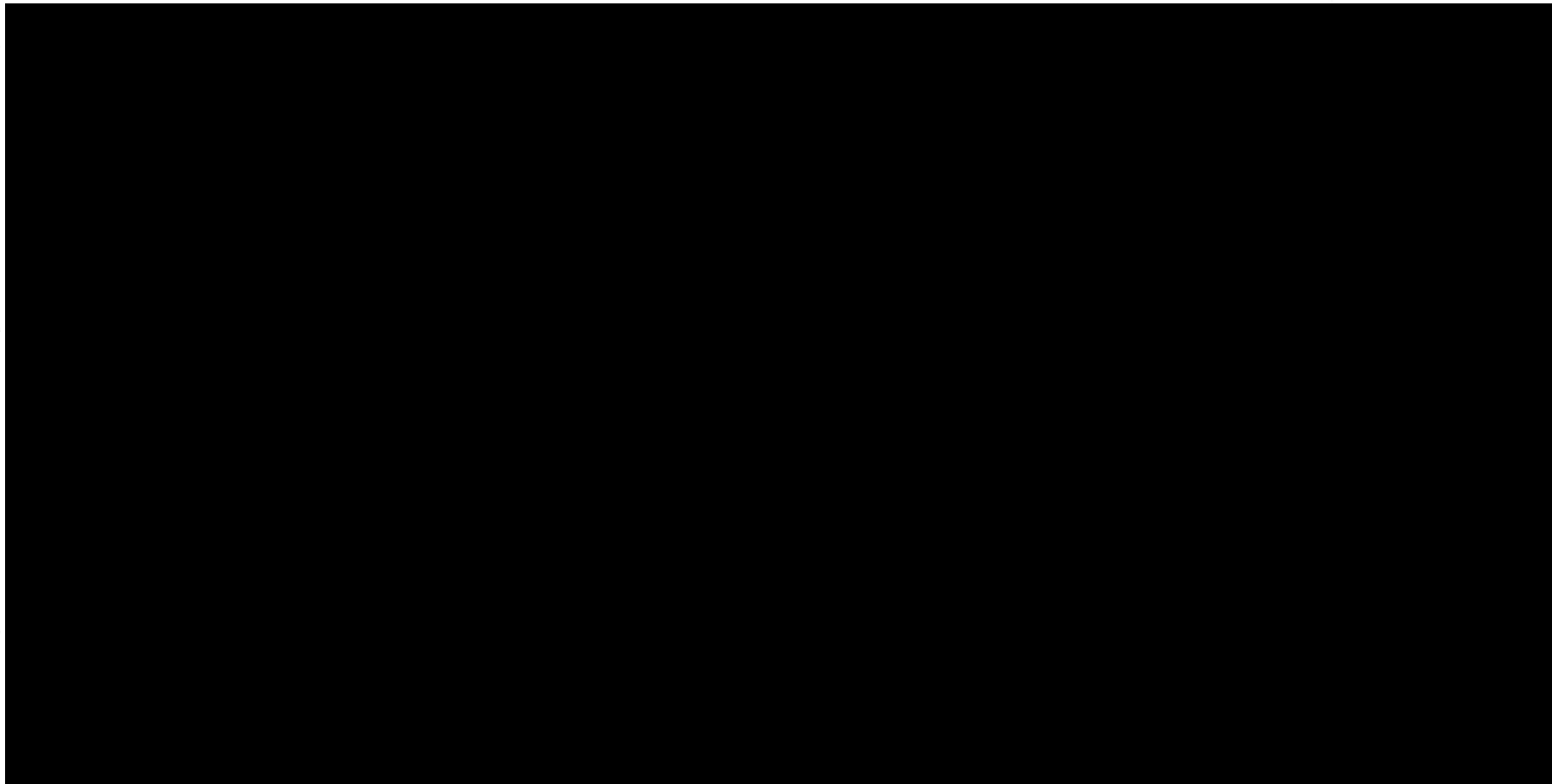
Instructions

1. Annotate each track plan in the way you think the editor should highlight the rule violation.

Think about:

- How to highlight the elements involved.
 - How to represent the specific nature of the rule.
2. Write down the textual description you think the editor should provide (feel free to take our suggestion as a starting point).
 3. Describe how you want the editor to display the textual description.

step 5
* Provide further
information through
link to document



D.5 Transcript of Plenary Discussion

Focus Group Transcript

Victor Cai

6 June 2022

1 General Workflow Feedback

Engineer 10: We were talking about single failures. It's very difficult to think about them in isolation, and there's an order we probably want to present the errors in to the user.

Some constraints are softer than others, e.g. balise required at a location vs. maximum separation between balises.

Engineer 8: User should be able to zoom in to the relevant part of the scheme plan from the error log.

Engineer 6: The order of presentation is important. For example, the 1500m separation rule should be at the bottom of the list.

Engineer 6: One thing I asked our group to start with which I didn't have the answer for was: who is the user? For example, if we have some balises in the wrong place, is the person working with the tool the one who decides where the balise should go, or are they just inputting the data (that someone else developed) into [redacted]? i.e. Is [redacted] part of the design process or is it just the second stage?

Engineer 15: First we ensure the positioning of BGs are compliant with the rules, then we look to optimize the locations.

Engineer 6: Do you design with [redacted] directly or do you start on paper?

Engineer 15: The old-fashioned way is to type in the locations into the [redacted] files. Nowadays we use a graphical tool at the beginning of the design process.

Engineer 6: So what you're showing to the user would perhaps be different?

[redacted]: [redacted] was done (drawn) separately in Excel. We tried to replicate this kind of functionality in the editor, so you can see on the screen where the elements are. My expectation would be that it (scheme plan design) goes straight in, along with the layout from the 3D scan of the railway. Then we add the ETCS equipment on top of that.

Engineer 6: Do you want the error messages at the time you're doing it [designing], or do you want them when you're finished, or do you want to be able to push a button to check an area?

Sometimes being told you are non-compliant when doing the design is really useful, but sometimes it's a right pain, because you haven't finished it yet!

- Don't tell me I've missed a balise when I've only just started doing it
- Tell me now that I've put one in a tuned zone which I'm not allowed to do

Engineer 15: Currently these checks don't run automatically. The user can choose to run the [redacted] and which rules to check, after doing some design.

[redacted]: Wouldn't you want to check by area? Or does it just check all of it?

[redacted]: At the moment it's just all of it.

[redacted]: That's probably something to think about, especially when it comes to scalability.

Engineer 6: You have to be careful not to overload the users with irrelevant error messages.

Engineer 14: That's why we're letting the user choose which rules to check, so that they can check the design in stages — starting with the most important rules, building up to eventually check everything.

2 Visualisation Presentations

2.1 Task 1

2.1.1 Group 1

Engineer 7: We show the distance as a dotted line, show the actual distance value, and the expected value in brackets. The error message can pretty much stay as it is, but we have a short and long version, where the long version also includes the track number and the positions of balises (you don't always need these details).

Engineer 8: The hexagons show the number of errors associated with each element.

Engineer 6: For the interaction, we can have a list of errors. The user can click on an error to zoom in to the relevant area.

"These balises are over 1500m apart, but you may also want to look at these other errors involving the same balises."

Markus: So instead of looking at the errors in isolation, you need to have some way of figuring out what the other errors are.

Victor: So, what do the rest of us think about their visualisation?

[redacted]: I really like the idea of showing the number of errors. It allows you to prioritise the elements with most errors.

Engineer 14: That's something I haven't thought about before. It's good to correlate the errors. I think it might be difficult to know.

[redacted]: We have presented the same in a different way. Perhaps once we've looked at the solutions from all three groups, then we can decide which of the three is the best.

Markus: Let's discuss the pros and cons after presenting all three solutions.

2.1.2 Group 2

[redacted]:

We highlighted the objects that were referred to by the error, i.e. the balises and the path between them. We also said it's useful to know the actual distance — if it's only a couple of metres out, we don't care.

Then we got into the discussion of how to display it. We assume there will be some kind of clickable error list. The highlight only appears when you click on / hover over the error. When you click on it, you also get the actual rule (verbatim text) and the full path.

2.1.3 Group 3 (5 steps)

Engineer 4: We took the idea that rather than having a different way of visualising each [type of] error, we have one way of visualising all the errors, with the caveat that we're looking at this from the perspective of a data entry person, at the end of the design workflow.

1. There are very few situations where there's only one error. Here, for example, there could be at least three. We indicate the area of the error using a box.
2. At this point, we're in the right ballpark, we want to look at the individual objects. We flash or highlight the elements to indicate that they have errors. It is then up to the user to decide which element to look at first. I think adding the number of errors on each element to our design would be useful.
3. We show a drop-down list of all the errors [on an element]. Once you select the error that you want to address (of course, a bit of thinking is needed to figure out the priority), irrelevant information is removed — these can be quite busy scheme plans, and if I put a highlight over a section, there could be multiple objects in that section that don't matter for the problem at hand; but maybe they do, which is why we say we remove the irrelevant stuff . . .

Engineer 6: We had that discussion. We thought that while automatic hiding is good, we also want the ability to bring them [the highlighting] back on again, because you need to have the whole picture.

Engineer 4: As a human being trying to read this though, you want to be able to limit that information to exactly what it is. So a "toggle display" function would be good.

4. The path related to the selected error is highlighted, along with a textual description.
5. We give the designer a link to the detailed information (e.g. design rule) rather than just a single line of text.

Markus: What would you like to do with our suggested textual description? Is it part of step 4, or where would you like to see it?

Engineer 4: In step 3, when you've selected the area, there should be a pop-up with the textual description as you hover over an error.

Engineer 9: Did the rest of you have any discussions about the sort of messages you present to the user?

There are two extremes: one is to just tell them what the error is; the other is to tell them what to do to fix the error. The danger with the latter is that the error may not actually be the problem. So how far back down that spectrum do you go? Do you just say that the distance is too large?

Engineer 6: I think that's where we ended up — just showing what the error is.

[redacted]: It can't be avoided trying to suggest a solution that's actually the problem.

Engineer 14: It comes down to how they come up with the rules in the first place, and behaviours ... What things have they considered in order to come up with this rule? To be able to come up with how to fix it, you have to take all that into consideration, and then provide an answer.

Engineer 6: The other thing is, some of these rules are *desirables*, as opposed to *must do it*. Therefore, it's an engineering compromise to decide which rules we have to respect, and which ones we're not too bothered about in the constraints we've got. You're never going to get a tool to make that sensible decision — it's actually going to mislead the human by giving them the obvious one, which may often be the wrong one!

Engineer 8: Some of the rules are marked as safety critical, so they must be the rules that addressing first.

Engineer 6: But in reality, for a VPCA for example, if we have a 1m tolerance, we just can't put a balise there. What are you going to do? Move the railway? No. You will take a tolerance, and decide on a different VPCA distance.

Engineer 4: In step 5, it may not tell you exactly how to fix the problem, but it describes in more detail what the problem is that you're trying to solve.

Markus: Coming back to your point that sometimes it's not possible to resolve things: in these cases, there could be some kind of "notice" where the person looking at the errors could say that they couldn't be resolved for the stated reasons.

Engineer 6: But for the one with the BG and the tuning unit, the answer may be that we need to move the tuning unit, not the balise group.

We have a short and long description, but we've also got a long list of errors. Your screen is only so big, so if you have a long description that takes up more than one line (it takes 2 lines), you'll only see half the number of things (errors). So we definitely want to have a short one to get the gist of it, and then we can see the details of the one we're interested in.

Engineer 4: So step 6 is the "marker" where we can say "I've looked at this, but I'm not going to fix it, hide it from now on". Here, we can either accept that it's okay, or defer it until we've fixed the other problems.

Engineer 6: Precisely. At least hide it for now, because otherwise you won't see the wood for the trees.

Engineer 9: So that'll be a requirement on how the [redacted] works, rather than the way in which the errors are displayed.

Engineer 14: The [redacted] has one for justifying why the rules are ...

[redacted]: ... Because when it finds an error, it might be the first such error that it's found and there might be more errors hiding behind that one. Because it's checking the constraints logically, the error you find might be the first instance of the constraint being violated, but it might not be the only one. Currently we only provide this first violation of a constraint for a particular balise.

Given two balises, there may be more than one way that the rule is violated, so it might justify one way but not the other.

[redacted]: If the user is allowed to hide the violation for now, then that rule can also be excluded from being sent to the [redacted] next time.

[redacted]: Yeah, you can just turn off the rule completely.

Engineer 0: What are you doing between b_1 and b_2 if there are two paths for the violation? Are you only showing one?

[redacted]: It'd only show you one at the moment.

2.1.4 Conclusion for Task 1

Markus: So what are the elements we find important, what are the things we find convincing, and why do we like/dislike the elements?

Engineer 14: For graphically showing the distance with the dotted line: it's great but I think it will be difficult. When you look at a particular plan, depending on how it's spaced out, it may not be easy to make sense of what is shown, because there's so much other stuff.

What we've seen when we try to display things is: trying to put labels on automatically makes it hard to be made readable. That's why I'd go for the simpler approach of highlighting the path, then show the measurements (distances) in the text.

[redacted]: Yeah, we already have a lot of problems with legibility and clutter.

Engineer 6: We were also thinking: when we highlight it, we may want to use a different colour for a missing balise as opposed to a superfluous one. It's a way to quickly see the nature of the error.

Engineer 7: I guess this example would be a "missing" colour.

Engineer 14: The problem I see with displaying the number of issues is: working out whether the problem is with the balise itself or with something associated with it.

Engineer 6: The error isn't necessarily to do with a single balise. In this example, we have two balises that are more than 1500m apart, so we need to move *one* of them. However, both balises are involved in the error.

Engineer 14: So there could be one nesting in the middle, from another rule. And that's the one you need to fix. Which balise do you assign the rule violation to in this case?

Engineer 4: It's not indicated which one is failing, but it is indicated which one's associated with it.

Engineer 6: The idea is: if you had a balise with only one violation (the distance constraint), and another one with 6 violations, then you should prioritise fixing the one with 6; don't bother with the balise with only that one violation, because you're almost certainly going to move the other one later.

Engineer 14: It just might be difficult to work out and get that correlation between ...

Markus: Well, after running the [redacted], you could simply say for example, balise 1 is occurring in 5 of the violations, balise 2 is in 7 of them, etc. Then you could simply display this number. It should be technically possible.

Engineer 10: I really like this idea, but I think I heard [redacted] say something earlier on, which if I understand correctly, undermines this.

In our example, both paths between the balises could be violating the rule, so the two balises will both be shown as having 2 violations. But I think [redacted] said earlier that it'll stop searching after finding the first one.

[redacted]: It'll check all the elements of the TDM, so if we're checking the balises, it'll check all the balises against each other, and all the balises against all the routes. If a particular pair of balises have two violations, then one of them will be hidden, but you'd know that the pair of balises violate that rule. In other words, you know which elements of the TDM are causing a violation.

Engineer 10: In this example, if the only problem is the two balises being too far apart (on both paths), how many errors would there be?

[redacted]: There would be one error.

Engineer 14: So in this case, we're just saying that they are too far apart, without further information.

[redacted]: We know they are too far apart because there is this path that's too long. The path is the witness.

Engineer 10: Do we know what the path is?

[redacted]: We do know what the path is.

Engineer 10: But there may be other paths which are also too long, which we won't find until we fix the first error ...

[redacted]: We could make the [redacted] search for more paths.

[redacted]: The [redacted] is trying to prove that there's an error. As soon as it finds an error, it stops.

Engineer 10: So if we fix the error, and we run the [redacted] again, it'd be the same number of errors?

Engineer 6: That sounds a bit counter-intuitive, doesn't it?

Engineer 0: If you put a balise on the bottom path, it fixes the error; but the next time you run it, you get a new error, because it'll find the (violating) path over the top branch.

Engineer 14: [redacted] you could get more, because you have placed it too near a set of points, for example.

Engineer 0: Indeed. If you put another balise down, [redacted] you accidentally put it in the tuning zone, you'd get more errors by "fixing" it.

Engineer 6: But at least you know it's something you've just done in that case, because that's true, you've created a new error.

Engineer 15: It's important to put the problem into context. It could be highlighting an area, a route, or other elements involved in the error. When you read the information provided with the error, you should know what [the nature of] the error is.

Engineer 14: If you're going to come up with a fix, you're going to need a lot more knowledge than what the tool tells you.

Engineer 15: It depends on the type of error. [...] Explaining the error in just one line of text can sometimes be more confusing than helping.

[redacted]: It's probably desirable to show all the errors at once, rather than showing one at a time. But if that's "hard" ...

Engineer 4: Then it goes on to [redacted], because there's *[inaudible]* there as well. You go on one bit, then another ...

[redacted]: But it is massively irritating.

Engineer 4: Oh yeah, absolutely.

[redacted]: It's something to bear in mind!

Engineer 14: With this example, if you didn't mention the path between them, you'd look at those balises, and wonder "how do I get between them?" But you might think of that anyway.

Engineer 4: I think what we're saying is that the context is more important than the number of errors. Not errors can be isolated to be on a single element. It may be good enough to just highlight the area (a higher level).

Engineer 15: It's always possible to just highlight the element, and show a table with the details (context). The choice of approach is up to you.

Engineer 4: There's a higher level above that though: the greater context of all the problems. If you're just doing one thing, then it's fine; but the problems can be related to each other.

Engineer 14: I think that's what we were saying about which one to switch on first — the rule in the example would probably be something you check towards the end, in which case you place a balise after each divergence of the things which you had to do to satisfy other rules.

[redacted]: So there's some prioritisation of rules that we need to do?

Engineer 4: Yes, I think so. Prioritise how you show them, certainly.

[redacted]: Currently there's just a field that says if a rule is safety critical or not. Maybe we want to have different levels of rules in the future.

Engineer 14: As you said, when they're constructing this data, they're going to run it [the [redacted] in stages; they are not going to construct all the data, then run it against the rules.

Engineer 4: It's not just in terms of the safety criticality, it's the *significance for needing re-design* that we really want to prioritise on. You obviously can't quite do that [right now], but yes you would look at things like repositioning balises after points, and where you put a balise in relation to an EoA, before you care about the maximum separation between balises.

2.2 Task 2

2.2.1 Group 1

Engineer 8: Here we highlighted the portion of the track where you shouldn't place a balise.

Engineer 6: Not quite what we're talking about here, but relevant: Each balise group has a dimension. Where we have the one dimension in [redacted], that's the "position 0" balise. Balise groups may have two or more balises. What we want here is *none of the balises in the group* being placed in the tuned zone, but we only have one position of where the first one is — it could be the leftmost or rightmost. So this rule in itself isn't good enough. We need to consider that the numbered (reference) balise could be outside the tuned zone, but some of the actual balises [in the group] could be inside.

Engineer 14: I think it's talking about balises, not balise groups?

Engineer 6: Yes, but don't we know where the balise group is?

Engineer 14: We know where every balise is. The balise group position is just the 0th element of it.

Engineer 6: Because all these balises are shown as one, I've assumed (because most BGs are two) that what we call a *balise* here is really a balise group.

Engineer 14: We show each individual balise.

Engineer 6: Right, so it may be how you've done your design there, but in reality most of these BGs should have at least two balises, because otherwise you don't have the directionality.

Engineer 8: So we show the exclusion zone highlighted in red.

[redacted]: Be careful with your choice of colours, because red and green have special meanings for the engineers ...

Engineer 8: Of course.

2.2.2 Group 2

[redacted]: We've done basically the same: highlight the area of the tuning unit, highlight the object that caused the problem, say what the actual distance is, and the error message, etc.

We had a bit of debate about whether it should be [inaudible] or not, but we figured it may be a bit hard to see if it is. Anyway, the user can always zoom in to see an object.

Engineer 14: Yeah, we also need to think about how much of the error to show at different zoom levels.

2.3 Task 3

2.3.1 Group 1

Engineer 7: We've decided to put a dashed icon in the location where the balise should be, and also change the wording from "to the left of" to "on the approach to".

Engineer 8: And a question mark to indicate something missing.

Engineer 9: This is because it might be the *VPCA start* that's incorrect.

Victor: You also said something about the directionality of nodes?

Engineer 6: Yes. What I was saying is that nodes like 3 and 7, which I'm thinking of as a route-setting point or EoA, I can't tell from the drawing which way the MA goes. You can infer it from the arrow on the *VPCA start*, but without that, we wouldn't have known.

Engineer 0: But if there'll be a signal ...

Engineer 6: Precisely. In that case you'd have a symbol that tells you the direction the signal applies to. A node doesn't tell me which direction the *VPCA* applies to. If I had a marker board, I would know.

These nodes are just names in the [redacted] model. They don't actually mean anything to a signalling engineer, who needs route-setting points.

Engineer 14: I think what's missing here are the entrance/exit markers when we show a route. We need to show them, rather than just highlighting the route.

Engineer 0: I think the problem with this example is: the *VPCA* is an offset from a marker or signal, which is not shown on the diagram.

Engineer 6: Precisely.

Engineer 10: In order to evaluate that error, we need more information than is shown at the moment. We need to know what that *VPCA start* is for.

2.3.2 Group 2

[redacted]: We have something very similar to show that a balise is expected there. We colour the balise in a way to show that it isn't actually there.

Markus: I think you also highlighted the *VPCA start*, that's different from the previous group.

Engineer 0: The problem is that the *VPCA start* isn't a physical object, but an offset from something (e.g. signal), so you probably need to show where that object is, so that the user can understand why the balise should be there.

2.4 Task 4

2.4.1 Group 1

Engineer 7: We thought it's better to show a balise missing than "being in the wrong place".

Engineer 6: We don't know if the balise on the red route is in the wrong place. You could have a balise there for many different reasons, that's nothing to do with this rule. So you can't say the balise is "wrong" — it may or may not be necessary. It's the one that's *missing* that we know is wrong.

2.4.2 Group 2

Engineer 8: You've also highlighted the *VPCA Start*.

Engineer 6: But I struggle with that, because if you have another rule saying you need a balise within 1m of the *VPCA Start* anyway, it could be on the other side of node 4 so it doesn't actually arise. And this is where a rule "overrides" another rule.

2.5 Task 5

2.5.1 Group 1

Engineer 6: We're talking about an "RBC configured route", but how can I tell from that diagram where the RBC configured routes are? I can't.

So I think if we knew where they were, we would highlight the route from the route-setting point to the end of the route, saying we are expecting to find a balise but we haven't. But as we said on the previous example, there isn't enough information on the plan to identify it.

Engineer 7: We use the "missing" colour to highlight the route, along with the "missing balise" icon.

Engineer 15: [Very long comment about the technical design of [redacted] and how the rules are high-level guidelines for working with the RBC compiler]

Engineer 9: I think I heard you say when we define an RBC session, we might want to get the RBC session fully defined, before we put down any RBC routes. But there's a rule that says the whole area of the session must be covered by RBC routes. Therefore, we can't check that we've defined the session in isolation from defining all routes, but it'd be really good if we could.

Engineer 15: Summary: At this point, the tool doesn't need to check that you've defined an RBC session, it just needs to check that you have an RBC route in each branch of the divergence.

Engineer 9: So you start by defining the areas, before you start defining the paths within the areas?

Engineer 15: It depends.

Engineer 14: The [redacted] is seeing if certain rules are infringed. The basic layout has to be valid, and the basic areas have to be valid, before we can get to this stage of processing. So I'm guessing if a bit of track wasn't joined, then the [redacted] must be able to say "this is an invalid layout" or "this is an invalid RBC area", because it uses those to check the rules.

[redacted]: The parser handles the building of the TDM and NEM. At the moment, if it can't build a valid TDM, it will stop working at that point.

Engineer 15: The tool is checking the relations between elements [...] At some point, you could fix an error and end up with 10 more.

Engineer 14: I think that brings us back to the point of prioritising the rules, and also introducing other ones that would help with that process. So we could define some rules to add into the [redacted] and other tools.

Engineer 15: A book [broad?] definition of the rules is mandatory. Then we need to assign priority to them to define the order of checking.

Engineer 14: From what we have so far, the user can select which rules to check. It requires some intervention to say "there's no point checking these until you've done these".

Engineer 15: It depends on the experience of the user. Someone who has worked with this kind of data might want some flexibility — they may want to see all the errors at once; whereas for a non-experienced user, we might want to tell them "you can't do that until you complete the first step".

[redacted]: I had this idea that we should have *layers* in the model — you start with the topology, then put in your positions, etc. Are you building up like that?

Markus: Summary of important points:

- Currently we consider the whole track plan, whereas in the future we'd like to be able to check just an area (fragment) of it.
- Rules need to be prioritised / bundled; the designer should be able to decide which rules are checked at each stage.
- In case of multiple violations, the user should get a list of (short) error messages in a window.

- (G3 step 1) When the user clicks on an error message, they should get a highlight of the relevant area.
- (Step 2) Highlight the objects; for each object involved, show the number of errors found in that check.
- (Step 3) For each object, show a list of violations.
- (Step 4) Show a detailed visualisation of the specific violation. Some elements could be hidden to make the highlighting better (optional feature)
- (Step 5) Reference link to the rule.

Does this interface sum up the ideas you brought to the table?

Engineer 6: We also need to be able to "ignore" an error. ([redacted] it as resolved)

Engineer 0: The list can be sorted by rule or by object.

Engineer 14: With the multi-step approach, I think highlighting the object and changing the zoom can be made a single step. If you do want to see the context, you could just zoom out.

Markus: When we design an interface, we need to consider how many clicks an action takes. Here we have a 4-5 step approach, which may not be ideal. This is something we need to reflect upon. But I like the systematic approach to it.

Engineer 6: We also need to give the user enough information so that they can relate what they see to their understanding of the real world. In my experience when designing control tables, the lack of contextual information makes it hard to tell apart different bits of the railway that look similar. I need to be able to relate the names in the computer abstraction to what is on a conventional scheme plan, including the layout direction.

3 Final Interaction Discussion

Engineer 6: It was difficult to get my head around the *totality* of the rules. You need to think about what rules are similar to each other, and what happens when you get violations of two of those rules.

Engineer 7: Classification of the rules is important, whether you are picking them or writing them, otherwise we can have trouble sorting them later.

Engineer 4: I think it'll be worth discussing the sequence of interaction in another workshop or study.

[redacted]: It's been very interesting and kickstarted some conversations we haven't had before.

Engineer 14: I quite liked the idea of filtering errors. The ability to group them is quite powerful.

[redacted]: One of the key problems of our collaboration with Swansea is: how do we transform the mathematical notation of formal methods into something we can explain to the end user? In the future, we hope to integrate Victor's extension into the editor ([redacted]).

Engineer 6: We need to remember that many infrastructure elements (e.g. AWS magnets) have dimensions themselves. Scheme plans are to be interpreted, but not literally.

Appendix E

Files Involved in the Extended Example

This appendix has been removed for intellectual property reasons.

