Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Energy

Hellebust, Haakon
Klevan, Sondre
Salihzada, Nima

Automated Railway Single-Line Diagram Generation Using RDS (ISO/IEC 81346)

Bachelor's thesis in Electrification and Digitalization
Supervisor: Frank Mauseth
Co-supervisor: Steinar Danielsen
May 2024

NTNU
Norwegian University of Science and Technology

Hellebust, Haakon
Klevan, Sondre
Salihzada, Nima

Automated Railway Single-Line Diagram Generation Using RDS (ISO/IEC 81346)

Bachelor's thesis in Electrification and Digitalization
Supervisor: Frank Mauseth
Co-supervisor: Steinar Danielsen
May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Energy

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis explores the creation of an automated system to generate railway single-line diagrams using the Reference Designation System (RDS) according to ISO/IEC 81346 standards. The project aims to develop a structured and efficient methodology for this automation, addressing the need for consistent and standardized documentation in the railway industry.

This project incorporates RDS principles and introduces standardized and hierarchically structured models, which aid in better management and understanding of railways' complex power supply systems.

Theoretical foundations include graph theory, utilizing concepts like directed acyclic graphs and trees to model relationships and dependencies within railway power supply systems. The study covers specific elements of Norwegian electrical railway systems, such as insulators, switches, motors, transformers, fuses, and loads. It also discusses the structuring principles, system levels, and reference designation needed for coherent and standardized documentation.

The methodology involves model development with multiple alternatives for implementing modeling principles within the RDS framework, along with detailed examples and implementation strategies. Database design focuses on creating relational databases to store and manage system data effectively, using primary and foreign keys, SQL, and the ltree extension. Algorithm development includes creating algorithms for tasks like topological sorting, visualizing connections, tracking component states, and implementing decision trees.

The results present several alternatives for RDS models, evaluating their effectiveness and applicability in the automated system. Insights into practical database implementation and integration with the automated system are provided. The performance of the developed algorithms in generating accurate and reliable single-line diagrams is also evaluated.

This thesis demonstrates the feasibility of automating the generation of single-line railway diagrams, which aligns with the emphasis on standardized documentation practices outlined in ISO/IEC 81346. With greater resources, it is possible to create a more extensive system to handle other types of systems.

Sammendrag

Denne oppgaven utforsker opprettelsen av et automatisert system for å generere enlinjeskjemaer for jernbane ved hjelp av referansebetegnelsessystemet (RDS) i henhold til ISO/IEC 81346-standardene. Prosjektet har som mål å utvikle en strukturert og effektiv metodikk for denne automatiseringen, og adresserer behovet for konsekvent og standardisert dokumentasjon i jernbaneindustrien.

Prosjektet inkorporerer RDS-prinsipper og introduserer standardiserte og hierarkisk strukturerte modeller, som bidrar til bedre håndtering og forståelse av jernbanens komplekse elektriske systemer.

Teoretiske grunnlag inkluderer grafteori, hvor konsepter som rettede asykliske grafer og trær brukes til å modellere relasjoner og avhengigheter innen jernbanesystemer. Studien dekker spesifikke elementer av norske elektriske jernbanesystemer, som isolatorer, brytere, motorer, transformatorer, sikringer og belastninger. Den diskuterer også struktureringsprinsipper, systemnivåer og referansebetegnelser som er nødvendige for sammenhengende og standardisert dokumentasjon.

Metodikken involverer modellutvikling med flere alternativer for implementering av tekniske systemdelingsprinsipper innenfor RDS-rammeverket, sammen med detaljerte eksempler og implementeringsstrategier. Databaseutformingen fokuserer på å opprette relasjonsdatabaser for effektiv lagring og håndtering av systemdata, ved bruk av primær- og fremmednøkler, SQL og ltree-utvidelsen. Algoritmeutviklingen inkluderer opprettelse av algoritmer for oppgaver som topologisk sortering, visualisering av forbindelser, sporing av komponenttilstander og implementering av beslutningstrær.

Resultatene presenterer flere alternativer for RDS-modeller, og vurderer deres effektivitet og anvendelighet i det automatiserte systemet. Innsikt i praktisk databaseimplementering og integrasjon med det automatiserte systemet gis. Ytelsen til de utviklede algoritmene i genereringen av nøyaktige og pålitelige enlinjeskjemaer evalueres også.

Denne oppgaven demonstrerer muligheten for å automatisere genereringen av enlinjeskjemaer for jernbane, noe som samsvarer med vektleggingen på standardiserte dokumentasjonspraksiser som beskrevet i ISO/IEC 81346. Med større ressurser kan man danne et mere omfattende system som da kan håndere andre typer systemer.

Acknowledgements

This thesis is the culmination of our work on developing an automated system for generating railway single-line diagrams using the Reference Designation System (RDS) according to ISO/IEC 81346 standards. It has been a challenging and rewarding journey, during which we have gained deep insights into the intricacies of electrical railway systems and the importance of standardized documentation.

We would like to express our sincere gratitude to our internal supervisor, Frank Mauseth, at NTNU. His guidance, support, and expertise have been invaluable throughout this project. His insights have contributed significantly to the successful completion of our thesis. And the pizza was delicious.

We also extend our heartfelt thanks to Steinar Danielsen, our external supervisor from BaneNOR. His assistance in developing the RDS model and his practical knowledge of railway systems have been crucial to our research. His support and encouragement have been instrumental in shaping the direction and outcomes of this thesis.

We are deeply appreciative of the time and effort both Frank Mauseth and Steinar Danielsen have dedicated to our project. Their contributions have not only helped us achieve our goals but have also enriched our understanding and appreciation of the field.

Thank you.

Sondre Klevan, Haakon Hellebust, and Nima Salihzada

Contents

List of Figures

List of Tables

Listings

Abbreviations

- ISO/IEC: Organizations that maintain terminological databases for use in standardization.
- RDS: Reference Designation System in accordance with ISO/IEC 81346 - 1, 2 and 10.
- RD: reference designation.
- DAG: Directed Acyclic Graph
- UID: Unique Identifier
- ERD: Entity Relationship Diagram

1 Introduction

Accurate and consistent documentation is essential for properly managing and running complex electrical systems in the ever-changing railway power supply systems field. The Reference Designation System (RDS), created in accordance with the ISO/IEC 81346 standard, meets this need. This project uses RDS to model the railway power supply system and automate single-line diagram generation. An automated system for single-line diagram generation would streamline the current process for engineers.

The project goal is tripartite:

1. Describe an electric railway system in RDS based on an electrotechnical understanding of power systems, facilities, and components.
2. Establish a relational database for the RDS description based on an understanding of information and data structures.
3. Show how the RDS description in the relational database can be used by the system owner based on, for example, programmed visualization and algorithms.

The theoretical foundation consists of a rudimentary explanation of electrical railway systems and RDS. Graph theory is also included, which is used to explain linkages and dependencies through structures like directed acyclic graphs and trees within railway systems.

The project's methodology includes database design, algorithm development, and model development. Detailed examples and implementation methodologies are shown, along with several options for applying structural concepts within the RDS framework. The fundamental goal of the database design is to efficiently store and manage system data by building relational databases with the help of SQL, the PostgreSQL ltree extension, primary and foreign keys. Developing algorithms involves implementing decision trees, visualizing data linkages, and sorting data topologically.

The results of this research present various alternatives for RDS models and trees, evaluating their effectiveness and applicability in the automated system. Practical insights into database implementation and integration with the automated system are provided, highlighting the performance of developed algorithms in generating accurate and reliable single-line diagrams.

2 Theoretical basis

2.1 Graphs

To understand RDS and the algorithms used in this project, it is important to possess a basic understanding of graphs and trees. Graphs consist of a set of *vertices*, also called *nodes*, and *edges*. In the graph, edges connect vertices. A vertex may hold information or have a label to distinguish it from other vertices. An edge is denoted by the two vertices that form the edge. For instance, if an edge exists between vertices u and v, the edge is denoted by (u, v). Vertex u and v are the *endpoints* of the edge and are considered adjacent vertices. Edges in a graph can be directed or undirected. If the edge (u, v) is directed, it is only possible to access node v from u, but not vice versa. However, if the edge is undirected, both nodes can access each other. This implies that (u, v) and (v, u) exist. Figure 2.1a) shows an undirected graph, while figure 2.1b) shows a directed graph.

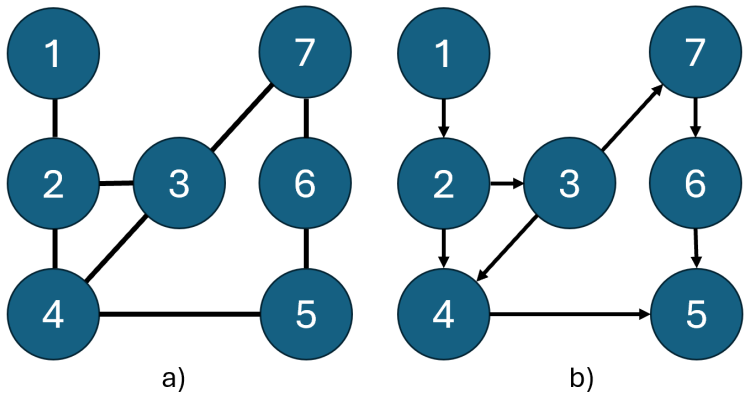


Figure 2.1: a) An undirected graph, b) a directed graph

2.1.1 Directed acyclic graphs and trees

A directed acyclic graph (DAG) is a graph characterized by directed edges and the absence of cycles. In other words, a DAG does not contain any paths where one can start and end up on the same vertex, distinguishing it from cyclic graphs [1].

Trees are abstract data types that represent hierarchies within graphs. Abstract data types are defined by their behavior, not by how they are implemented. Trees are often implemented as a list of nodes. The implementation must also contain information about the edges, provided in the form of adjacency lists or matrices. These are two-dimensional data structures because in a DAG with n vertices; for any vertex u, there can be up to $n - 1$ edges with u as the source. If the DAG allows for self-loops, edges that travel from a vertex to itself, each vertex can have n edges. In any case, DAGs can have up to n^2 edges, explaining the edge representation data structure. This concrete concept is used for most other types of directed graphs.

Rather than being defined by the implementation, trees are defined by their behavior seen from the user's point of view. One of the behaviors that define trees is the notion of "parents," "siblings," and "children." In trees, nodes can only have one parent, but they can have multiple children and siblings. If a node has a parent, the parent node is one step higher in the hierarchy and on the same branch of the tree as its child nodes. Meanwhile, sibling nodes share parents [2]. Figure 2.2 visualizes a simple tree structure.

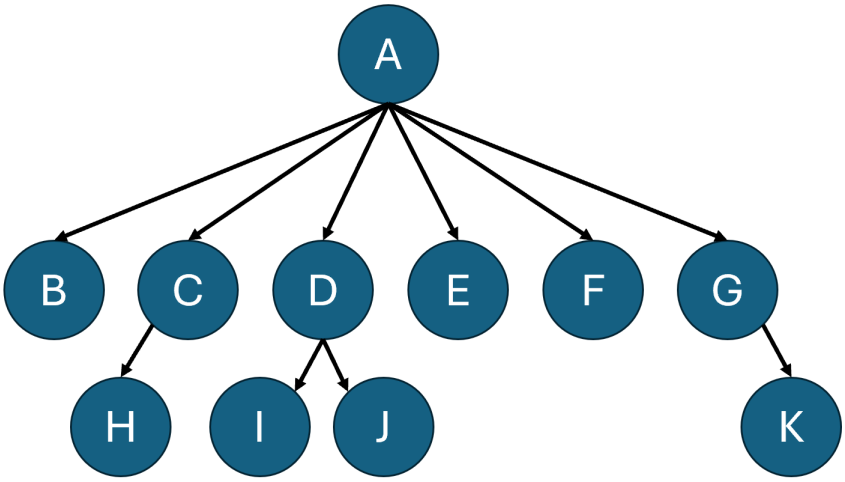


Figure 2.2: Example of a N-ary tree

While regular graphs often do not care about how the graph is oriented in space, trees are frequently presented in layers. Figure 2.2 shows a tree where node *A* is the *root* node. Root nodes do not have any parent nodes and are at the top level of the tree hierarchy, meaning that they are *ancestors* to all the other nodes in the tree. This notion makes sense as all nodes in the tree can be traced back to the root node by following the arrows reversely. All of these tree-specific behaviors define tree structures, and the behaviors are indeed abstract. Trees can appear as undirected graphs, but only trees with directed edges will be considered in this project.

Trees can optionally have guidelines on how many children each parent node may have. A binary tree, for instance, only allows nodes to have at maximum two children. This project only uses *N-ary* trees or *generic* trees, which are trees that allow any amount of children per parent. A *subtree* is a commonly used term defined as a tree that is a node's child. For instance, from the root's left child's perspective in a binary tree, it is the root of a *subtree* consisting of all its *descendants*.

Classifying nodes in terms of how many children they have is frequently helpful. The *degree* of a node is the total amount of children the node has. A node with a degree of zero, a node without children, is referred to as a *leaf* node, while nodes with higher degrees are considered *internal* nodes.

2.1.2 Height-balanced trees

Another unique property of the tree structure is *height*. The height of a vertex is the longest downward path to a leaf from that vertex. The height of the tree is then the height of the root. Height is a property that comes in handy when a tree's *balance* becomes important. A completely balanced tree is one in which the height of the left and right subtree for any node does not differ by more than one. Figure 2.3a) displays a balanced binary tree. Figure 2.3b) shows an unbalanced tree, where from the root's perspective, its left subtree has a height of 1, while its right subtree has a height of 3.

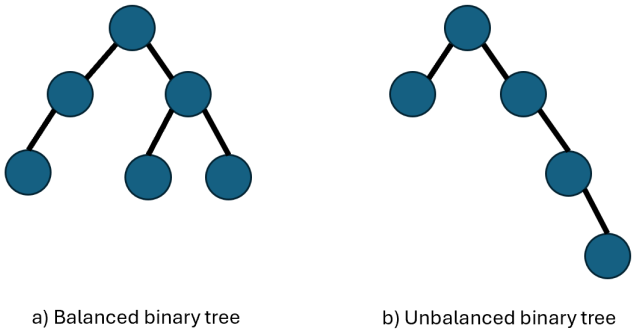


Figure 2.3: a) a balanced tree, b) an unbalanced tree

2.2 Electrical systems

This section will provide a basic overview of the electrical systems and components encountered in this project.

2.2.1 Norwegian electrical railway systems

Norwegian railway power supply systems use catenary systems to supply the trains with electricity. Using pantographs, the trains connect to the *overhead lines*. The lines are maintained at a voltage of 15kV and a frequency of 16.7Hz. When the train contacts, the current flows through the motor and the wheels before following the return circuit through the tracks and to the line [3].

2.2.2 Single-line diagrams

Single-line diagrams are simplified representations of electrical systems. Lines in the diagrams connect *nodes* that represent electrical components within the system. Figure 2.4 shows a single-line diagram of a station. The station has two tracks with sectioning overlaps on both sides. Switches are in parallel with the sectioning overlaps. A switch is connected to track 1, leading to a fuse, a transformer, and a load.

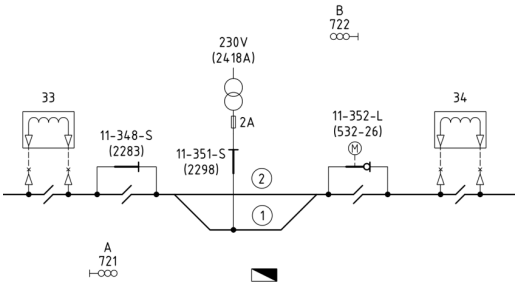


Figure 2.4: Example of single-line diagram

An overview of all the components encountered in this project can be seen in figure 2.5. The following sections cover the functionality of each component.

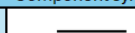
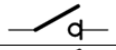
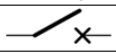
Component	Component symbol	Component	Component symbol
Line/Conductor		Disconnect switch	
Sectioning overlap		Load breaker	
Section insulator		Circuit breaker	
Fuse		Motor	
Transformer		Booster transformer	

Figure 2.5: Component name and component symbols

2.2.3 Insulators

Two different insulators, or sections, are encountered in the single-line diagrams in this project: *sectioning overlaps* and *section insulators*. Sectioning overlap is a physical configuration of the line, whereas a section insulator is a component. Both represent short stretches where the train makes no contact with the line. The primary function of section insulators is to divide electrical transmission systems in two. Trains can pass through sections with their pantographs raised.

2.2.4 Switches

Railway power supply systems use three switches: *disconnectors* (disconnecting switches), *load breakers*, and *circuit breakers*. The disconnector is a mechanical device that can either be in an open or a closed position. In a closed position, the current flows through as normal. In an open position, no current can flow through, and the disconnector fulfills the requirement for an isolation function. The isolation function is the ability to "cut off the supply from all or a discrete section of the installation" [4]. Disconnectors cannot open with current flowing through, meaning another component is needed to break the current for the disconnector to open.

Load breakers, often called load switches, are mechanical switching devices capable of carrying and breaking currents under normal conditions [4]. The load breaker can also break the current in specific overload conditions.

Circuit breakers are only found within the converter stations in the railway systems in this project. "Circuit breakers are mechanical switching devices, capable carrying and breaking currents" [4]. Compared to a load breaker, the circuit breaker can also break high short-circuit currents. Even though circuit breakers have better safety features, they are not always used due to their high cost. One circuit breaker is also sufficient for breaking the current for the entire system, meaning only one is needed to prevent damage from short-circuit currents.

2.2.5 Motors

A motor's functionality in the railway is to enable remote-controlled switches.

2.2.6 Transformers

Two types of transformers appear along the railway. One of them is the regular transformer, which is "a device that transfers electric energy from one alternating-current circuit to one or more other circuits, either increasing (stepping up) or reducing (stepping down) the voltage" [5].

The other type of transformer is the booster transformer. A booster transformer's task is to ensure the voltage level is correct [6]. A booster transformer has a winding ratio 1:1. Booster transformers help guide the return current to the return circuit as described in section 2.2.1 [7].

2.2.7 Fuses

Fuses are safety components. They are placed where it is most critical to mitigate damage. Fuses are made to handle specific currents. If the current exceeds this value, it will break, leading to no current flow. A fuse can only break the current once and needs to be physically replaced after current interruption [8].

2.2.8 Loads

The railway system has termination lines, also called loads. In the single-line diagram, these are represented as arrows or lines that lead nowhere. If the termination line is preceded by a transformer that transforms the voltage to a level below $1kV$, it is considered a low-voltage termination. Otherwise, it is considered a high-voltage termination.

2.3 RDS

RDS provides a set of principles for structuring systems and is the common formula for standardized letter codes for any technical object. By correctly implementing these principles, databases describing large systems can be handled efficiently. Effective utilization of these principles requires knowledge of certain RDS concepts.

2.3.1 Object

An *object* is defined as an entity involved in a process of development, implementation, usage, or disposal. An object does not need to exist physically. "It is the designer/engineer who decides that an object exists and establishes the need to identify this object" [9]. For instance, an autonomous robot is built on physical parts, but software is an equally important part of the robot. While intangible due to its nonphysical nature, computer software is still considered an object.

2.3.2 Aspect

It is often useful to look at objects from different angles. These are referred to as *aspects*. Aspects essentially filter objects and highlight the relevant information. RDS provides four aspects: the product, location, function, and type aspect. The product aspect highlights constructional relations (assembly) of the object's components, while the location aspect highlights the spatial relations between components [9]. These aspects have their use cases but are not used in this project. However, the function and type aspects will be used extensively. Figure 2.6 illustrates how an object viewed from different angles shows different aspects.

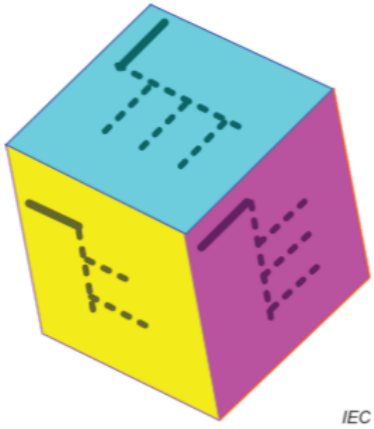


Figure 2.6: Object represented as a cube with a different aspect on each side [9]

2.3.2.1 Function

A system's purpose is to execute a process. In the context of RDS, a *function* refers to the task of an object within such a process without considering its implementation [9]. Because this project considers railway power supply systems, the function of the system as a whole is the transmission of electrical energy. Therefore, the function aspect is modeled with this specific function in mind.

2.3.2.2 Type

A type in RDS is a grouping of objects with a particular set of characteristics in common. Depending on the number of common characteristics, types can vary from very generic to very specific [9]. Consider, for instance, power supply systems. It could be helpful to define different types for different power supply system voltage levels: low voltage, medium voltage, and high voltage. These types are relatively general but are still unambiguous, which makes them valuable types. This example is visualized in figure 2.7.

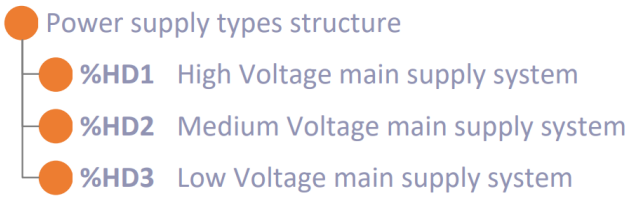


Figure 2.7: An example of the type aspect [10]

The type aspect is often combined with other aspects, such as the function aspect. When an object is referred to in the function aspect, the type aspect provides specificity.

2.3.3 System

A *system* is defined as a set of interrelated objects considered in a defined context as a whole and separated from their environment. Systems are generally determined by the definite function they perform or the given objective they attempt to achieve. A system is considered an object, and the objects that make up the system are its *sub-objects* [9].

2.3.4 Structuring

In order for a system to be efficiently specified and designed, the systems are normally divided into parts. Each of these parts can be further divided. This successive subdivision into parts and the organization of those parts is called *structuring* [9]. In this project, the structuring will be based on the function aspect.

Such a structure divides a system into constituent objects in the function aspect based on a top-down approach. In a top-down method, the process is:

- 1) Select an object
- 2) Determine its sub-objects

This process is repeated for each object in the system.

2.3.5 System levels

Systems can be separated into three *levels*: power supply, technical, and component systems.

2.3.5.1 Power supply systems

Power supply systems are "systems used to transmit, convert, distribute, and store energy". Power supply systems may recursively incorporate other power supply systems as one of their elements[11].

2.3.5.2 Technical systems

Technical systems are those systems that "are not by themselves considered to be power supply systems but represent technical solutions useful for the realization of a power supply system." They may incorporate other technical systems recursively as one of their elements [11].

2.3.5.3 Component systems

Component systems encapsulate the individual components, encompassing both basic elements and component assemblies. Component systems may incorporate other component systems recursively as one of its elements [11].

2.3.6 Reference designation

Reference designations, or RDs for short, aim to unambiguously identify objects within systems. RDs can be *multi-level* or *single-level* [9].

2.3.6.1 Single-level reference designation

A single-level RD should include a letter code, a number, and a prefix. RDS-2 [12] provides a set of stable class codes to identify different component systems, whilst RDS-10 [11] provides class codes for power supply and technical systems. Numbers follow the letter codes to distinguish object occurrences. The class codes for power supply, technical, and component systems are of length one, two, and three, respectively. For instance, if two objects of the same class exist in a system, they would be distinguished by unique numbers. A prefix should also be included to clarify what aspect is in use. Table 2.1 shows the different prefixes and their implementation.

Table 2.1: Different aspects in RDS

Aspect	Prefix	Example
Function	=	=A1
Product	-	-A1
Location	+	+A1
Type	%	%A1

2.3.6.2 Multi-level reference designation

A multi-level RD consists of two or more objects. The objects may be from any of the three system levels. The objects are separated by a dot. A dot represents the relationship "belongs to", reading from right to left. In figure 2.8, object B belongs to object A, and object E belongs to B. The multi-level RD in the function aspect for E is then =A.B.E .

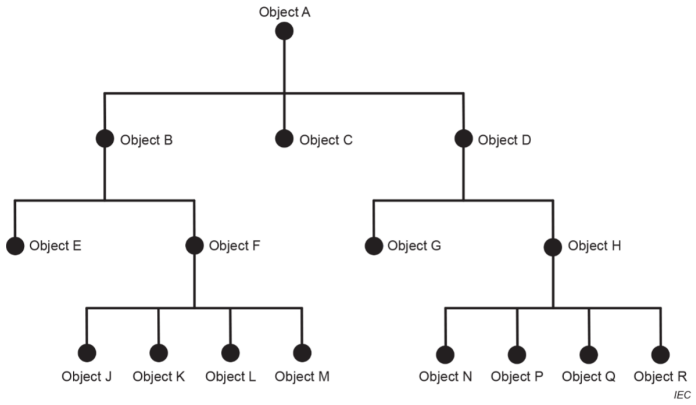


Figure 2.8: Tree structure of object A [9]

Collectively, multi-level RDs of all objects in a system form a tree structure. In tree structures such as the one shown in figure 2.8, object A is the *top node*, which is the tree's root that represents the system. Its descendants represent its sub-objects [9].

To keep the trees comprehensible, the power supply systems should have at most twenty-five children combined, where the children are power supply, technical, and component systems.

2.3.7 Top nodes

The top node is used as a label and describes what system the RDS tree models. The top node is denoted by <>. For instance, if one were to model a national railway system, the top node should identify the system and describe what part of RDS is used. A top node of the following structure is common: <BaneNOR.A-B.RDS-PS>, where PS denotes RDS-10 (Power Supply Plants), and A and B denotes locations.

2.3.8 Categorization

RDS includes class codes for a vast array of electrical systems. However, table 2.2 shows only the class codes relevant to this project.

Table 2.2: RDS codes used for the railway

RDS code	Description
J	Electrical energy guiding system
B	Electrical flow control system
JE	System for transfer of electrical power
KF	Transforming system
KL	System for control of electrical energy flow
WBC	High voltage wire
WBA	High voltage busbar
UAA	Insulator
QBA	Disconnector or load breaker (switch)
QAB	Circuit breaker
XBA	High voltage termination
XDA	Low voltage termination
MAA	Motor
TAA	Transformer
FCA	Fuse

2.4 RDS structuring principles

To achieve consistent results, a set of principles for structuring with RDS is necessary. The principles used in this project have been developed through internal discussion within the team.

Firstly, the power supply systems are to be separated by circuit breakers and open disconnectors. Secondly, component systems are separated by function. Technical systems, however, can be divided in many different ways. *Linking systems* intend to link two other systems on the same hierarchical level. They are, therefore, often used to separate technical systems in the model. This section explores different ways to structure these linking systems.

2.4.1 Overlapping and edge-to-edge technical systems

An overlapping configuration allows technical systems to overlap, resulting in fewer technical systems. Even though the tree is more concise, the technical systems contain few details, making it difficult to recreate the original system from the tree.

Edge-to-edge does not allow for technical system overlapping. This means that each technical system has unique sub-objects, thus making the tree more structured and detailed, though larger. It is important to strike a balance between conciseness and detail. Concise trees are smaller and easier to parse but may need more information, while detailed trees may get too big, harming their readability. Overlapping and edge-to-edge configurations are visualized in figure 2.9.

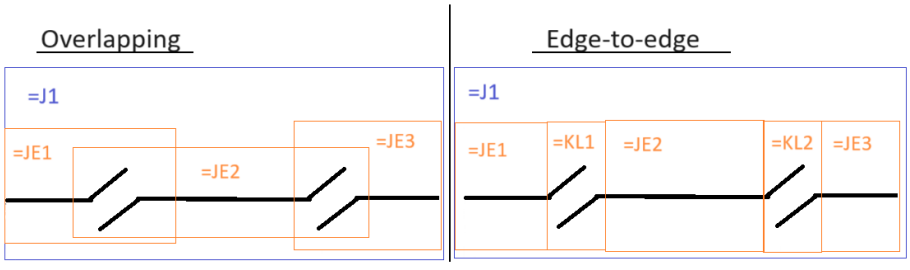


Figure 2.9: Overlapping vs. Edge-to-edge RDS

2.4.2 Receiver ownership principal

The *receiver ownership principle* clarifies which technical systems linking systems should be part of. Where a system is intended to link two other systems on the technical system level, and when it is unclear to which system it belongs, the linking system should be part of the receiving system. For instance, a valve controlling flow from a water tank to a tube would belong to the tube. This is due to the direction the water flows. Figure 2.10 exemplifies this.

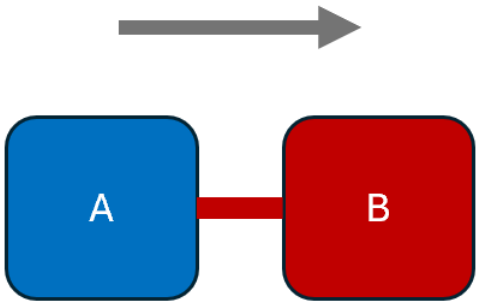


Figure 2.10: An example of the receiver ownership principle. The linking system between A and B belongs to B

2.4.3 Station-oriented structuring

Station-oriented structuring separates technical systems based on the occurrence of a railway station. A KL system encases the entire station and the linking systems on each side. Figure 2.11 shows how this principle is applied to a sample system.

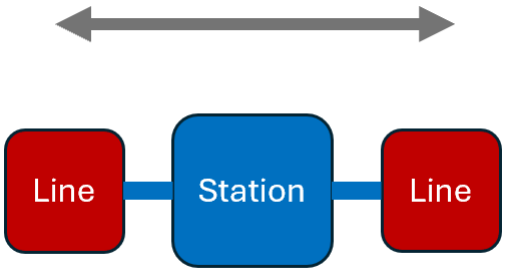


Figure 2.11: An example of a station-oriented structuring. The blue box and the blue links between the boxes are subsystems of a KL system

2.5 Databases

A database is an organized collection of data. Databases are accessed through database management systems, which consist of integrated computer software that allows clients to interact with the database. This project uses the relational database PostgreSQL.

2.5.1 Relational databases

A relational database is a type of database that stores data in tables with rows and columns. In general, each table represents one *entity type*. Consequently, each row in the table represents an instance of that entity type, while the columns represent values attributed to that type. The database is *relational* in the sense that entities consist of attributes related to each other [13]. This

database paradigm is the most popular and widely used, making it easy to integrate with other technologies as many third-party software exists for most use cases. A table could, for instance, define three columns: `id` of type `number`, `name` of type `text`, and `date` of type `date`. A valid row must contain a valid ID, name, and date to be a valid row in the table.

2.5.1.1 Primary keys

Primary keys are minimal sets of attributes that uniquely specify a row in a table. When a new row is added to a table, a new primary key is produced with a unique value. The primary keys in this project are integers. In PostgreSQL, the program automatically increments the integer value of each new row so the primary keys remain unique for each row.

2.5.1.2 Foreign keys

Foreign key columns allow table rows to be coupled with rows in other tables. A foreign key column points to a primary key column in another table. Primary keys must be unique identifiers so that each primary key referenced by a foreign key refers to only one row. The table containing the foreign keys is considered the child table, while the referenced table is the parent table. This is a hierarchical model where if some referred-to data in the parent table is deleted, rows in the child table may now be invalid as the reference does not exist anymore. This logic does not apply the other way, as the parent table's attributes are independent of the child table.

2.5.1.3 SQL

SQL (Structured Query Language) is a domain-specific language for managing data in a relational database. The language is split into several distinct parts: *SQL schema statements*, which are used to define the data structures stored in the database; *SQL data statements*, which are used to manipulate the SQL schema statements, and *SQL transaction statements*, which are used to begin, end, and roll back transactions[14]. Transaction statements will not be used in this project.

For instance, if a client intends to create a new table in the database, the SQL schema statement `CREATE TABLE` should be used. However, if the goal is to populate a table, the SQL data statement `INSERT` should be used. Although SQL schema statements are essential, they are generally not used often as creating tables is frequently only done once per table. Data statements, however, are used constantly as the data rows inside the tables change constantly. Retrieving data from the database is also a part of SQL's data portion and is performed with a `SELECT` statement.

2.5.1.4 ltree

`ltree` (label tree) is a PostgreSQL data type for representing labels of data stored in a hierarchical tree-like structure. A *label* is a sequence of symbols, while a *label path* is a sequence of labels separated by dots. `J1.JE2.WBC1` is, for instance, a valid label path. As the label format harmonizes with RDS multi-level RD tags, and ltree creates a hierarchical tree-like structure, ltree was deemed an appropriate data type for this project [15].

2.6 Graph algorithms

The database contains relevant information about the system's objects. However, the data may not be structured appropriately. Since this project adopts foreign keys to facilitate relationships between tables and relationships can be represented as graphs, powerful graph algorithms improve the workflow.

2.6.1 DFS

Depth-first search (DFS) is a graph traversal algorithm that searches *deeper* in the graph whenever possible. The algorithm starts at a source node and explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all edges leaving v have been explored, the algorithm *backtracks* to traverse edges leaving the vertex from which v was discovered. This procedure is repeated until all vertices reachable from the source are discovered [2].

2.6.2 Edge classification

When DFS is run on a graph, it becomes possible to *classify* its edges. Analyzing edges in a DAG can be useful in determining certain graph properties. DAGs can have four types of edges: tree edges, forward edges, back edges, and cross edges.

- If the algorithm visits an undiscovered node v from a node u, then (u, v) is a tree edge.
- A forward edge is an edge that links a node to a descendant that has already been visited.
- A back edge is an edge that links a node to an ancestor that has already been visited.
- A cross edge connects two nodes that don't share any ancestor-descendant relation.

Edges are classified into these four categories during DFS's runtime. Back edges are often the most critical to identify as they indicate the existence of graph cycles. Forward and cross edges can also provide useful information about the graph's structure. For instance, if any back, cross, or forward edges exist in the graph, the graph can not be a tree, as trees only consist of tree edges.

2.6.3 Topological sorting

A topological sort of a graph is a linear ordering of all its vertices such that if the graph contains an edge (u, v), then u appears before v in the ordering. Such an ordering is only possible for directed acyclic graphs (DAG) as cycles would cause circular dependencies [2]. Figure 2.12 displays how a topological sorting algorithm works on a DAG. The input for the algorithm is a list of nodes supplemented by an adjacency list or matrix. The node list will be reordered appropriately by the end of the algorithm's runtime.

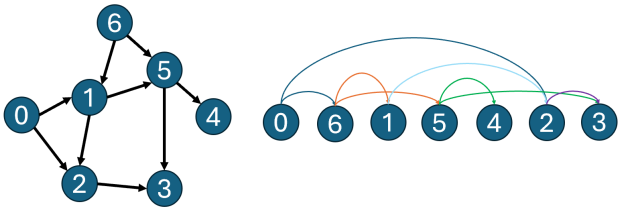


Figure 2.12: Illustration of a topological sorting algorithm

2.7 Software terminology

2.7.1 Separation of concerns

Separation of concerns is a design principle that divides computer programs into distinct sections. Each section is intended to address separate concerns. One way to apply this principle is to modularize code such that related code is grouped into separate modules. This approach makes programs easier for humans to parse and the software to develop in the long run.

Another separation of concerns in software development is the separation of concerns between the presentation layer, and the data access layer. These are often referred to as the *front-end* and the *back-end*. The presentation layer is part of the OSI model of computer networking, which is a model developed by the International Organization of Standardization (ISO) in 1984. The model allows for the separation of concerns within network communications. The presentation layer's job is to translate network data into presentable data for different applications [16]. In this project, this layer will be referred to as the front-end. The back-ends of software are the programs that process data behind the scenes and attempt to provide relevant information to the front-end. A rule of thumb is that the front-end is any component manipulated by the user, while the back-end usually resides on the server, often far removed from the user. The front-end and back-end are also often referred to as client-side and server-side.

2.7.2 API

An Application Programming Interface, or API for short, is a software interface that allows two computer programs to communicate. In most applications, multiple programs are used to fulfill various duties. Still, unless the different programs have built-in functionality to connect them, it can be useful to construct programs that provide such interfaces. For instance, if databases are used with other programs, APIs are usually created to retrieve data from the databases, connecting the different programs in the process.

In most cases, APIs are developed as server-side web APIs. A server-side web API is an interface that exposes one or more *endpoints*. These endpoints are hosted on a URL via a web server so that clients can access them by visiting the appropriate URLs. When a client visits the URL, an HTTP request is sent to the API and processed by an API endpoint *handler* function. After processing the request, the handler function returns an HTTP response to the client. If all the necessary API endpoints are designed properly, the client-side and server-side can then be considered to be *connected* through the API.

2.8 Programming technology

2.8.1 JSON

JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent syntax for defining data interchange formats. The format is inspired by object literals from ECMAScript, a scripting standard that intends to ensure web pages' interoperability across different browsers. JavaScript, which will be introduced in the next section, is also based on this standard, making all JSON code valid JavaScript code. This fact makes it easy to work with JSON in JavaScript, but JSON is still designed to be easy to use with other languages [17]. Many languages, including those used in this project, require some JSON parsing framework to convert JSON to valid code, but this process is often simple. Due to the efforts put into making JSON language-independent, it is often used as the data format when data is interchanged between software. In this project, JSON will be used when data is transferred between the database and the front-end through the API [17].

2.8.2 JavaScript

JavaScript, or JS for short, is a programming language and an important part of the Web. JavaScript makes it possible for developers to create dynamic and interactive web pages that can interact with visitors. Currently, 99% of websites use JavaScript for client-side website behavior [18]. It is possible to use other programming languages for front-end development. Still, since JavaScript is as widespread as it is, the JavaScript front-end ecosystem is vaster than any other, making development more efficient.

2.8.2.1 p5.js

p5.js is a front-end JavaScript drawing library based on the free graphics library *Processing*. p5.js is simply a JavaScript port from this library. Processing is known for being a user-friendly drawing software, which makes it easy to program visual designs. p5's built-in tools are also used to access external API endpoints and load JSON. p5.js is often used for rendering purposes but can also be used to draw stationary images. The built-in *draw* function is used to render frames, while the built-in *setup* function can be used to only draw objects once.

2.8.2.2 D3.js

D3, short for Data-Driven Documents, is a JavaScript library that allows for dynamic and interactive data visualizations in web browsers. It supplies the user with building blocks for creating customized data visualizations. With this library, one can target specific elements of the data input. Simple commands can be used for visualization when using preprocessed data.

2.8.2.3 Node.js

Node.js is a free and open-source server-side JavaScript runtime. The platform makes it possible to run JavaScript on the back end. This can be used to build server-side applications like web servers and APIs. Before Node.js and other frameworks appeared, web development relied on other languages for the back-end. Node.js makes it possible to use only JavaScript for both, making development more practical and fast.

Node is not used as the back-end implementation for the APIs in this project, but it is used to host the front-end working directory on the web locally. The *browser-view* node package (installed via Node Package Manager (NPM)) is used for hosting on localhost, making it possible to access all files of the chosen directory. Images and other data that exist in the directory can then be loaded from the JavaScript front-end dynamically.

```
1     browser-sync start --server --directory --files "*"
```

This command, when run in the command line of the desired directory, serves the entire directory and all the files in it on localhost with the standard port being port 3000. In the case of this project, it serves all the prerequisite p5.js library files, the required utility files, and the implementation files.

2.8.2.4 JSDoc

JavaScript is a weakly typed language. Weakly typed languages have less strict typing rules at compile time compared to strongly typed languages. These two typing paradigms both have their advantages and disadvantages. Weakly typed languages usually make for swifter and more efficient development, but it is easy to make simple mistakes related to variable types. Strongly typed languages ensure that if, for instance, an integer-typed variable is assigned to a string, the program crashes at compile-time which guarantees that no type errors make it to production. This comes at the cost of development speed, but in many use cases, type safety is a requirement.

Even though JavaScript is a weakly typed language it is possible to make the development of JavaScript programs more type-safe. JSDoc provides a way to declare types, and document behavior in JavaScript files. JSDoc does not ensure type safety as it does not interact with the JavaScript interpreter. It is rather used to help developers understand their own and other developers' work. The following code snippet is an example of how to use JSDoc to document a function:

```
1 /**
2  * Takes a full path and returns the component system label in the path
3  * @example "RDS.J1.WBC1" -> "WBC1"
```

```
 * @param {string} path
 * @returns {string}
 */
function getComponent(path) {
      let pathArray = path.split('.')
      return pathArray.pop()
}
```

Listing 1: An example of how JSDoc can document a function

A JSDoc comment is declared with the `/** */` syntax, where the documentation appears in between the brackets and stars. This particular documentation states that the *path* parameter should be a string type. It also affirms that the function returns a string. In addition to these type assertions, an example is provided that shows how the function works. When clients intend to use this function, they will have all this information available in their code editor, and the variables in use will have type annotations.

In addition to functionality documentation, JSDoc provides its own type definition system. Using the `@typedef` tag makes it possible to declare custom types. In the context of this project, it is, for instance, useful to define a custom Component type. The Component type is a JavaScript object with three properties: an ID (number), a Path (string), and a Type (string). The syntax for this is visualized in listing 2.

```
/**
 * @typedef {object} Component
 * @property {number} ID
 * @property {string} Path
 * @property {string} Type
 */
```

Listing 2: An example of how JSDoc can define custom types

2.8.3 Go

Go is a statically typed programming language designed by Google and released in 2009. The language takes inspiration from many older and established languages, especially languages such as C++. It is a compiled language, meaning Go code is translated to native machine code before the program is executed. This methodology differs from languages such as Python and JavaScript. These languages use interpreters, which execute the code step-by-step at runtime. Compiled programs are typically much faster to execute since the translation process of the interpreted languages introduces more overhead.

Go was created as a primarily server-side programming language. The language even includes an HTTP package in its standard library making it simple to initialize APIs and to handle requests from clients. Python was initially used for the back-end API for this project, but as other languages were explored, Go appeared as a solid alternative due to its speed and API expertise. Contrary to the Python API implementation, the Go API barely relies on third-party packages or libraries. The only dependency is a PostgreSQL driver package called **pgx**. This package contains PostgreSQL-specific functions used to initialize and close connections to the database, and querying the database for information.

3 Method

3.1 Model development

It is essential to develop an appropriate model for the system before it is used for software development purposes. Suppose an inappropriate or inaccurate model is used for the programming part. In that case, it is easy to pursue wrong patterns and ideas, often leading to frequent refactoring of the code, making the development less efficient. Consequently, modeling was the main focus for the first part of this project. The model structure underwent several phases as different RDS model alternatives were proposed.

3.1.1 Technical system division principle implementations

All subsequent alternatives use the edge-to-edge configuration, as defined in section 2.4.1.

3.1.1.1 Alternative 1

The first proposed structuring alternative, alternative 1, divides technical systems based on linking systems (switching systems). Linking systems are encased by KL systems. The remaining systems are classified as JE and KF systems depending on function. This principle ensures that two JE and KF technical systems are never adjacent. Alternative 1 separates the system into a granular network of technical systems, as shown in figure 3.1. However, the power supply systems in the RDS tree end up with too many children, according to section 2.3.6.2. This alternative was, therefore, quickly discarded.

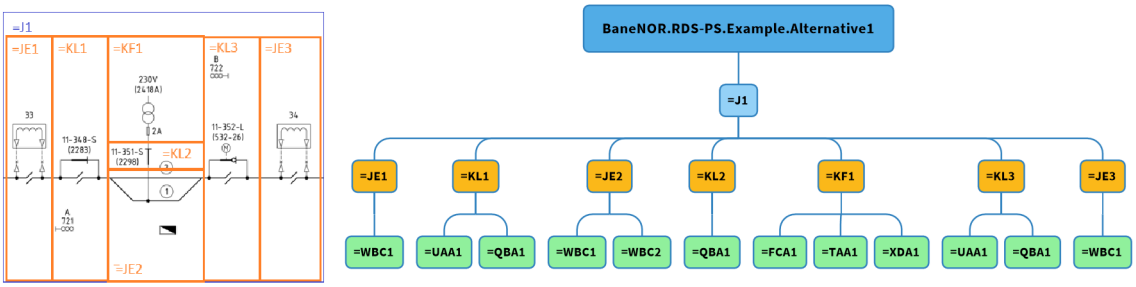


Figure 3.1: Alternative 1 implementation. The component systems are added in the tree though they are not modeled in the implementation

3.1.1.2 Alternative 2

Alternative 2 incorporates two technical systems, KL and JE, and is station-oriented. In this alternative, switches in combination with sectioning overlaps are considered the linking systems. Accordingly, stations and linking systems are encompassed by KL systems. The technical systems between stations transport electrical energy between them and are, therefore, JE systems. Figure 3.2 shows how this alternative is implemented on a sample system.

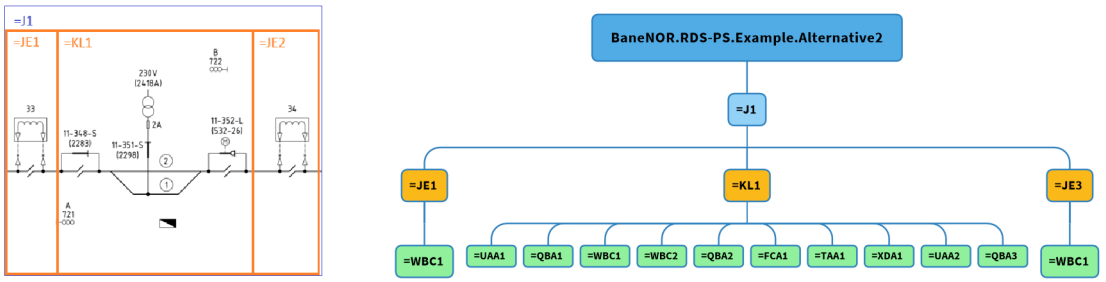


Figure 3.2: Alternative 2 implementation. TThe component systems are added in the tree though they are not modeled in the implementation

Alternative 2 causes fewer technical systems for each power supply system. However, the KL systems consist of many component systems, making the technical system level less detailed.

3.1.1.3 Alternative 3

Unlike alternative 2, alternative 3 delves deeper into station descriptions by introducing subsidiary technical systems. The linking systems on either side of the station can be split into subsidiary technical systems, KL1 and KL2, as shown in figure 3.3. This, in turn, causes fewer component systems per technical system.

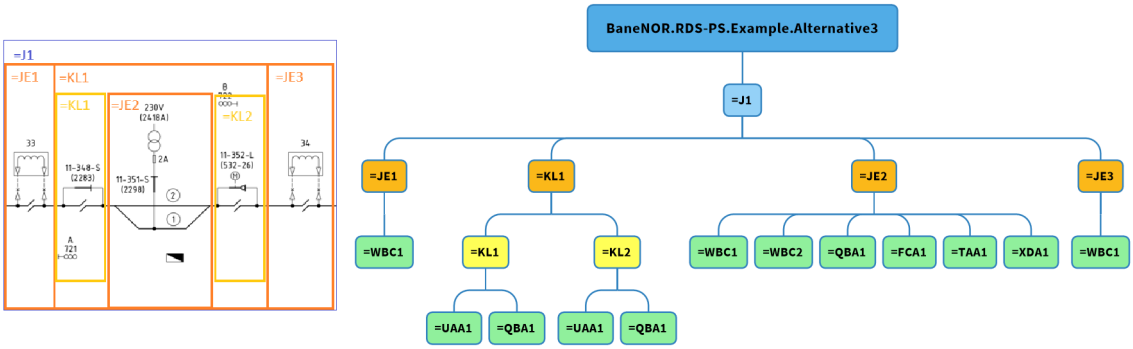


Figure 3.3: Alternative 3 implementation. The component systems are added in the tree though they are not modeled in the implementation

In this alternative, the system between the subsidiary KL systems is considered a JE technical system. This system is not a subsidiary of the main KL system; they are on the same hierarchical level.

3.1.1.4 Alternative 4

Alternative 4 improves on alternative 3 by turning the JE system into a subsidiary of the main KL system. Thereby, stations and linking systems are completely encased by a KL system. Additionally, the stations are divided into subsidiary systems, providing more detail. Alternative 4 blends positive qualities from alternatives 2 and 3, resulting in an optimized structure. Consequently, alternative 4 was chosen as the model structure for the rest of the thesis.

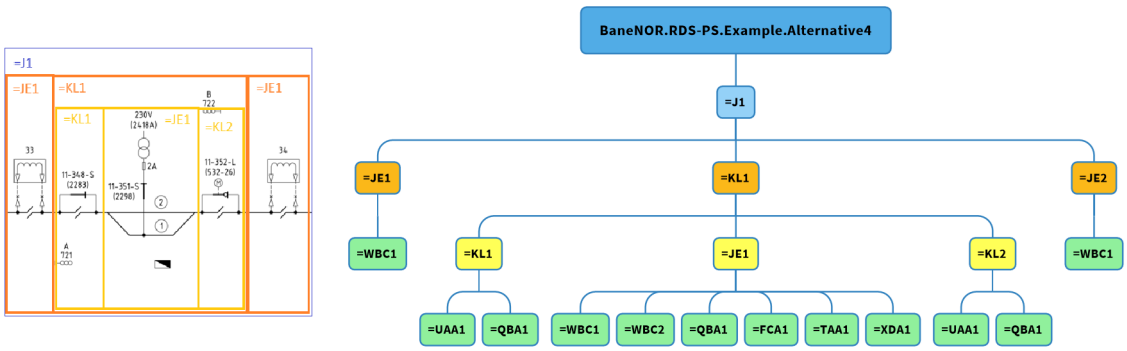


Figure 3.4: Alternative 4 implementation. The component systems are added in the tree though they are not modeled in the implementation

3.1.2 Example RDS implementation

This section shows how to apply alternative 4 to the example system in figure 3.5.

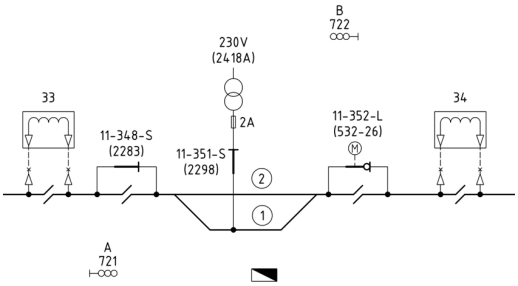


Figure 3.5: Example system

Firstly, one must identify the power supply systems. As the system does not contain circuit breakers in combination with open disconnectors, only one power supply system is present. The system's function is to transmit electrical energy and is, therefore, an electrical energy guiding system (J). The only RD from the first step of the implementation is `=J1`, as visualized in figure 3.6.

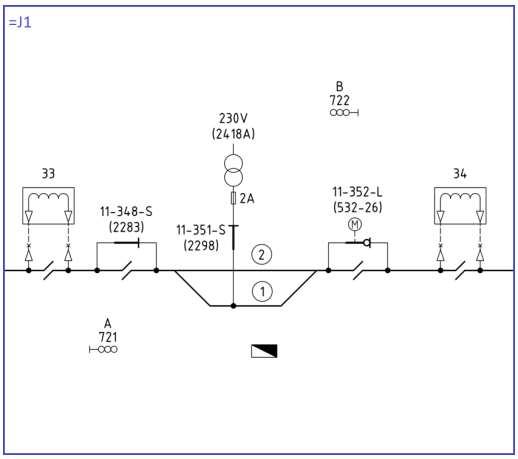


Figure 3.6: Example system modeled with power supply system

Secondly, technical systems must be separated according to alternative 4's principles. As the system contains a station, a KL system should enclose it, according to the station-oriented structuring principle. The switches and insulators on both station sides are part of the KL system. The rest are JE systems, as no other stations or switches exist. This produces the multi-level RDs: `=J1.JE1, =J1.KL1 and =J1.JE2`.

The KL system is further divided into subsidiary technical systems depending on functionality. The sectioning overlap and switch combinations control electrical flow. Consequently, they comprise KL systems. The system bounded by these KL systems is a JE system since its function is to transport electrical energy. This produces the following multi-level RDs: `=J1.KL1.KL1`, `=J1.KL1.JE1 and =J1.KL1.KL2`. The implementation of the technical system is visualized in figure 3.7.

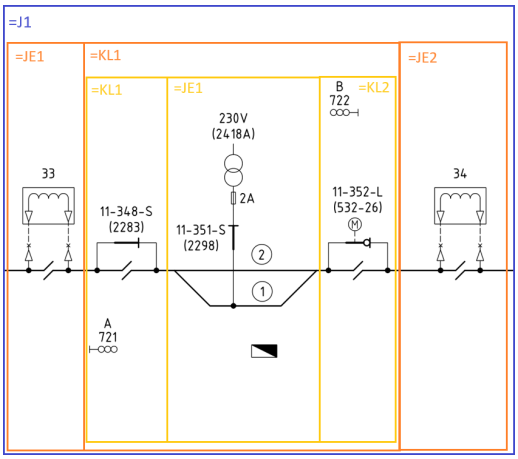


Figure 3.7: Example system modeled with power supply system and technical systems

The next step is to identify the component systems. This process is straightforward and consists of separating components based on their functionality. Starting with JE1 and JE2, the sectioning overlap and booster transformer are modeled as a single WBC component system, as the booster transformer connects both sides of the sectioning overlap. This produces the following multi-level RDs: `=J1.JE1.WBC1 and =J1.JE2.WBC1`.

Inside the subsidiary KL systems, switches and sectioning overlaps constitute the component systems QBA and UAA, respectively. Inside JE, the transformer, fuse, switch, and two parallel station lines comprise the TAA, FCA, QBA, and WBC component systems, respectively. Therefore, the following multi-level RDs are created: `=J1.KL1.KL1.QBA1, =J1.KL1.KL1.UAA1, =J1.KL1.KL2.QBA1, =J1.KL1.KL2.UAA1, =J1.KL1.JE1.WBC1, =J1.KL1.JE1.WBC2,`

`=J1.KL1.JE1.QBA1, =J1.KL1.JE1.FCA1, and =J1.KL1.JE1.TAA1`. The transformer transforms the voltage down to 230V and leads to a load. Therefore, the terminated line after the transformer is classified as a low-voltage termination (XDA), resulting in the multi-level RD `=J1.KL1.JE1.XDA1`. The implementation of the component systems is visualized in figure 3.8.

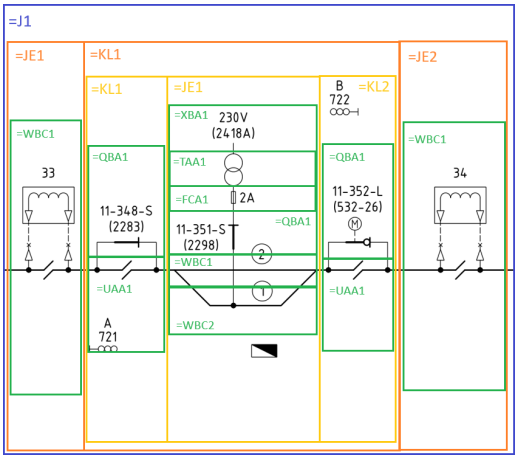


Figure 3.8: Example of station with power supply, technical, and component system

The final step of the implementation is to make a top node. This is done according to the principals given in 2.3.7. For this example, the top node can be <ExampleRailway.RDS-PS>.

The multi-level RDs can be visualized in a tree structure, shown in figure 3.9.

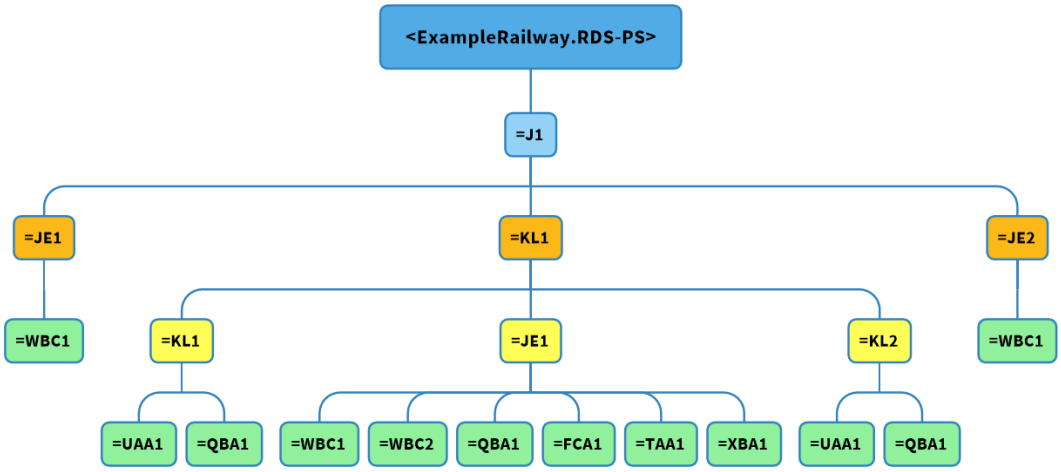


Figure 3.9: RDS tree from the example

This example is simple, consisting of just a single station. In this project, larger and more complex systems are encountered. However, the principles remain the same.

3.1.3 Implementation of the type aspect

Currently, the component system class codes RDS provides are often imprecise in the function aspect. Load breakers and disconnecters are, for instance, both classified as QBA, even though their functions differ. Hence, the type aspect is useful as it distinguishes component systems with identical class codes in the function aspect.

The type aspect can also distinguish systems that perform the same function but are different physical products. Sectioning overlaps and section insulators, for instance, are identical functionally but are different physical products. More types were added to the model during the development process. These will be presented in subsequent sections. All types introduced in this project are shown in section 4.4.

3.2 Database Design

After the model was developed, a simple database was created to fit the model to data points. The initial database contained one object table. This table's rows were populated with multi-level RDs and unique identifiers (UID). RDs are implemented as `ltree` data types. The UIDs are primary keys and get incremented automatically in PostgreSQL for each object, so the IDs stay unique.

The database structure needed to be expanded to capture relational data within the system. As this project is specifically concerned with the function of electrical power transmission, a new table was created to represent electrically connected components: the `connection` table. A row in the connection table consists of three columns: two foreign keys referencing objects and one UID.

The objects' multi-level RDs changed as the model changed during the project's course. Therefore, the rows in the tables were often altered in response. The table structures, however, remained the same until the type aspect was introduced. The type aspect was implemented in the database to provide further information about objects. This can be represented as a one-to-many relation, where an object can only have one type, but a type can be associated with multiple objects. Consequently, a new table was made to hold different types, and a column was added to the object table with a foreign key pointing to the type table. Figure 3.10 visualizes the database in an Entity-Relationship Diagram (ERD).

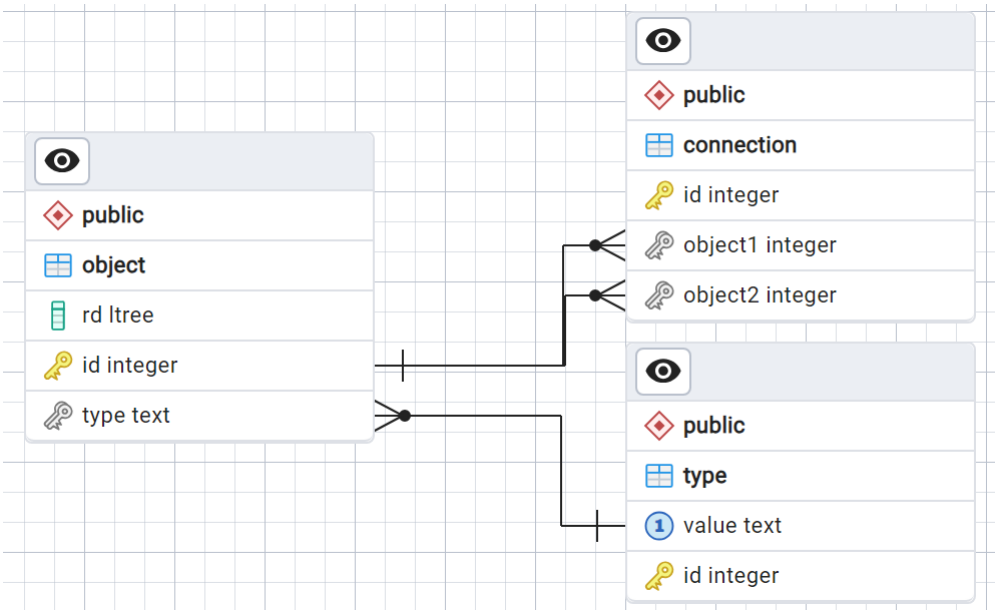


Figure 3.10: ERD (Entity-Relationship Diagram) for the database

3.3 Algorithm development

The programming section of the project consists of two main components: the back-end and the front-end. The back-end operates on the server side and provides relevant data to the front end, which is where the actual drawing takes place.

3.3.1 Initial algorithm experimentation

The initial stages of the coding section involved the development of a preliminary drawing algorithm. Instead of using data points from a database, this program simulated realistic electrical components expressed as custom-made JavaScript objects. As described in section 2.8.2.1, p5.js generally uses functions to draw various shapes. Initially, this project used an object-oriented programming style where class structures were used to encapsulate different components' properties.

For each component class code (e.g., WBC, UAA, QBC), a class was designed with a corresponding function that draws the component to the canvas. This is a good start, but how do the class objects know where to draw the components? Each class structure contained a constructor function that accepted two input coordinates, x and y. These coordinates represented the previous component's *connection point*. When a component was drawn to the canvas, the object calculated a connection point, which subsequent components treated as starting points.

The algorithm's foundational concept is to loop through a list and draw components to the canvas based on the information in the current iteration. The first version of the algorithm looped through a list of components, drawing the present iteration's component to the canvas reliant on the last component's connection point. However, this assumes that the components' order is such that they can be drawn successively and produce a coherent electrical single-line diagram as a result. In most cases, this does not apply. However, the central concept of the algorithm can remain the same if sufficient groundwork is performed on the back-end before the data is transferred to the front-end. An API makes this possible.

3.3.2 API construction

Constructing an API was vital as it provided a way to connect the database to the front-end. APIs can also manipulate data so that the front end receives well-structured data. This version of the API was developed with Python and provided two API endpoints: objects and connections.

The object endpoint is quite simple as the object data in the database is already structured ideally for the front-end use cases. This endpoint has, therefore, remained the same during the entire development process. The HTTP handler function for objects first connects to the PostgreSQL database using Pythons *psycopg2* package, a PostgreSQL adapter for Python, making it easy to interact with databases. Secondly, the handler function sends a query to the database asking for all rows from the object table. This query is a simple SQL command:

```
1    SELECT * FROM object
```

The symbol "*" means *all*, so this command selects all columns from the object table, i.e., the ID and RD columns. At this point, the table rows have been assigned to a variable in the Python program. This data is not appropriately structured. Therefore, before the data is returned to the client, the handler function initializes an array, loops through the variable containing the rows, creates an object for each component containing its ID and RD, and appends the objects onto the array. At this point, the data is structured well, so it is sent to the client.

The connection endpoint has almost the same logic as the object endpoint. However, instead of fetching from the object table, it fetches from the connection table. Another difference is that the ID column in the connection table does not provide useful information for the client. The SQL query is changed accordingly:

```
1    SELECT object1, object2 FROM connection
```

where `object1` and `object2` refer to IDs from the object table via foreign key constraints. The same procedure used in the object endpoint converts the fetched data to an ideal structure.

3.3.3 Front-end structure development

Now that the API is up and running, the front-end must find a way to communicate with it. p5.js has a built-in function called `loadJSON`, which can asynchronously access a file or URL and return a JSON object. In this case, the function sends a request to an API endpoint URL and awaits a response. When a JSON object is returned, it is assigned to a variable. There must be two `loadJSON` calls at the start of the program: one for the objects and one for the connections. These functions are called in the built-in `preload` function. If asynchronous function calls are placed in the `preload` function, the drawing functions and other logical segments will wait for these asynchronous calls to finish.

Since the front-end consists of a data loading section, object class data structures, and the main drawing algorithm, separating it into multiple modules was deemed appropriate. The main module would contain data fetching and the drawing algorithm. These two functionalities must be in the same module since the algorithm uses the fetched data. The secondary module contains the necessary data structures used by the main module. This modularization of code follows the separation of concerns design principle as defined in section 2.7.1.

At this point, the entire project was well-structured, and the front-end had a fully-fledged connection with the back-end. For the rest of the project, the algorithm could use realistic data points from the database when drawing.

3.3.4 Visualizing connections as a graph

Because the algorithm takes a relatively extensive dataset as its input, it became progressively more challenging to maintain an accurate mental image of the graph. Therefore, some effort was directed toward creating a graph visualization tool. Such a visualization tool would be beneficial when developing the algorithm, and it would also be possible to compare the drawn diagram with the graph to spot mistakes in the data. Open-source graph visualization tools are abundant, so constructing a new graph visualization algorithm would be a waste of time.

D3 has easy-to-use algorithms for trees, but these algorithms were quickly discarded as the input graph may not always be a tree. Although the graph ideally is highly hierarchical, multiple root nodes, cross edges, and forward edges may exist. The visualization algorithm should, therefore, be able to handle any DAG. A possible option was to use a force-directed graph drawing algorithm. Such an algorithm attempts to draw a graph where the edges are of more or less equal length, with minimal overlap. This is achieved by assigning forces among the set of edges and the set of nodes, based on their relative positions, and then using these forces either to simulate the edges and nodes' motion or minimize their energy [19]. They are also designed to be aesthetically pleasing.

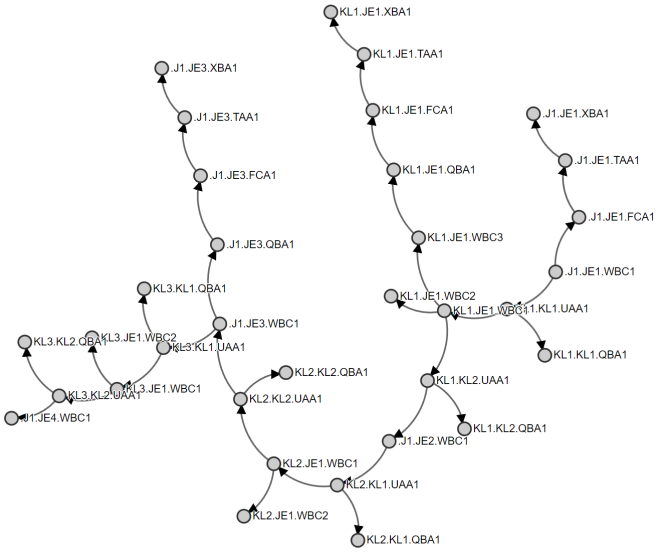


Figure 3.11: A visual representation of the connection graph using a force-direct graph drawing algorithm. The labels are shorter than their actual RD to improve readability

The graph in figure 3.11 represents the system and its structure well. Directional arrows clarify the order in which the algorithm is supposed to draw the components. They also indicate dependencies between objects. For instance, a connection from an object u to another object v indicates that to draw object v, u must already be drawn.

3.3.5 Implementation of the topological sorting algorithm

The concept of the algorithm in the first place was to draw every component iteratively in some sequence. Therefore, a function must exist to order the list of connections appropriately. One of the ideas that stuck out was to use a topological sorting algorithm. As explained in section 2.6.3, a topological sorting algorithm orders an input graph topologically. As connections are expressed as directed edges in a graph of components, the topological sorting algorithm finds an order of the components where for every directed edge (u, v), u comes before v in the ordering.

This helps the drawing algorithm as it makes it possible always to draw the components that other components depend on first. Since topological sorting algorithms are non-deterministic, as explained in 2.6.3, the resulting list of connections may differ on each iteration. Source nodes are nodes that do not depend on any other node. The topological sorting algorithm does not care which of these nodes comes first in the ordering. This fact leads to some level of unpredictability. However, the order will still be valid in terms of dependencies.

Because the front-end's only concern is to draw objects to the canvas, it only wants the connections in a topologically sorted order. It was, therefore, natural to implement the topological sort inside the API, hiding the implementation from the client. Hiding the implementation and only providing the necessary data to the client is an often-used design pattern for APIs. When a request is received for connections, the API fetches the data from the connection table, transforms the data to a graph representation, runs topological sort on the graph, and returns a topologically sorted list of edges to the client. The fetched connection rows are scanned and attached to objects of the custom type *Connection*. This type, as shown in figure 3.12, stores two integers named `Object1` and `Object2` of type `int64`, a type capable of storing integers ranging from -2^{63} to $2^{63} - 1$.

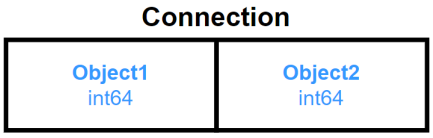


Figure 3.12: The connection type defined in the Go back-end

The topological sorting algorithm function implemented in Go takes a list of connection objects as input. Subsequently, a graph is initialized based on the connections provided. The graph is implemented as a *map* data structure. Maps consist of key and value pairs. They *map* keys, which must be unique, to values that do not need to be unique. In this map, object IDs are keys, and neighbor object IDs are values. Figure 3.13 shows an example of a map. A client can then, for instance, request the values associated with key 1, and get $[2, 3]$ in response. Internally, Go maps are implemented as *hash maps*. Hash maps use hash functions to make sure operations are faster. Figure 3.13 is enough to understand the behavior of maps in most cases. As the figure suggests, a key's value can be empty.

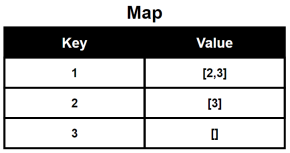


Figure 3.13: A simple Go map that maps integers to lists of integers

For each connection, if `Object1` does not exist in the map, it is added to the map with an associated empty list as its value. If `Object1` exists in the map, `Object2` is appended to its list. This creates an adjacency list graph representation. Additionally, an `in-degree` map is initialized to track how many edges are directed at each vertex in the graph. The vertices with no incoming edges are subsequently appended to a queue. The algorithm then starts a loop that terminates when the queue length is zero. For each iteration, the first node is removed from the queue before the program iterates over all the node's neighbors. For each neighbor, the edge between the dequeued node and the neighbor is appended to the sorted list, and the in-degree of the neighbor is decremented by one. Lastly, the algorithm checks if the neighbor has an in-degree of zero, and if that is the case, it is appended to the queue. This approach successfully sorts graphs topologically, as edges are only appended to the sorted list if the source node does not depend on any other nodes.

3.3.6 Tracking component state

The algorithm has limited context about the other drawn components to this point. Looping through the connections and drawing the target object solely based on the source object's coordinates works for trivial systems, but this approach is not scalable. The scalability of the software, although not a significant concern in this project, should be serviceable. The resulting structure of the software should be suitable for further development. Therefore, it seemed necessary to overhaul the algorithm's core concepts. Ideally, the algorithm should be versatile such that systems of slightly different structures do not leave the algorithm bewildered.

Consequently, more weight was placed on using already-drawn components to dictate the drawing locations to make the algorithm more flexible. This idea was leveraged as a data structure in the program. A JavaScript array, `drawnComponents`, was declared in the program. This array's job is to store information about all the currently drawn components. A new JavaScript class called `ComponentState` was defined to encapsulate this information. This class, as shown in figure π, has a constructor function that initializes four member variables: `x` of type `number`, `y` of type `number`, `id` of type `string`, and `type` of type `string`. The `x` and `y` variables make up a point on

the two-dimensional canvas representing the component's connection point. Although this can not replace drawing based on accurate location data such as GPS data, it is a solid solution for this project as exact positioning is not required, and only relatively simple systems are considered. The `id` and `type` variables offer some context so that the next component in the loop can draw differently depending on the preceding component's type. `drawnComponents` is then a list containing `ComponentState` objects.

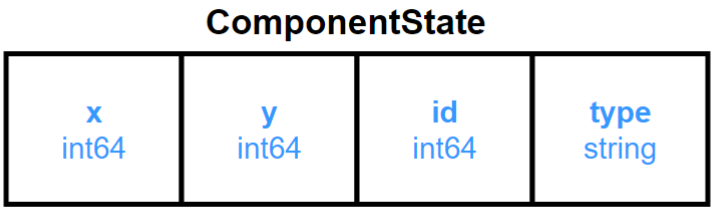


Figure 3.14: The ComponentState class as defined in the JavaScript front-end

A helper function `findComponentState` was defined to facilitate efficient usage of the list of drawn components. This function takes two parameters: an ID and a list of drawn components. The function loops through the list of drawn components, checking if the current iteration's component matches the input ID. If a match occurs, the component state is returned from the function. Conversely, if there is no match, `null` is returned.

3.3.7 Decision trees

As the program iterates through the connections, it uses the `findComponentState` function to check if the source component's state matches any component in the list of drawn components. If a matching component state is found, the algorithm proceeds to determine the appropriate drawing function for the specific context by evaluating a series of logical conditions. *Decision trees* offer a useful mental model for understanding this process. A decision tree, illustrated in figure 3.15, outlines the various paths the algorithm can take from the starting point of each loop iteration [20]. In this example, each square, circle, and diamond represents a JavaScript function. When the algorithm encounters a logical condition, the decision tree branches out into different paths, exploring various functions.

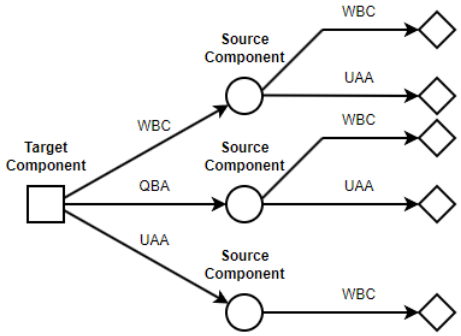


Figure 3.15: An example decision tree visualizing the algorithm's decision structure. Labels on arrows leaving the square represent possible target components, while arrows leaving the circles represent possible source components. The diamond symbols represent the possible states the algorithm can end up in

For instance, if the algorithm identifies the target component as a switch (QBA), it must determine the source component class. Conditional statements guide the traversal of the tree based on this information. Suppose the source component is an insulator (UAA). In that case, the algorithm will invoke a function that draws switches in relation to insulators, indicated by the diamond shapes in the figure. All decision paths end in functions that draw a specific component in a particular context. This methodology ensures reasonable drawing accuracy, as most components must be represented differently depending on the context.

3.3.8 Type aspect integration

The type aspect excels at providing additional context where context is lacking. When integrated into the software, the type aspect can be considered an additional layer in the decision tree. That is, it causes further branching in the tree and, consequently, more accurate component drawing. Initially, types were introduced to differentiate between component system variants that have the same class code but look different in single-line diagrams. Table 3.1 shows the types implemented for this reason. When all QBA and UAA types are implemented in the database, the algorithm traverses differently in the decision tree depending on the target component type.

Table 3.1: Different component system types

Switches	Type
%QBA1	Disconnector
%QBA2	Load breaker
%QBA3	Remote controlled disconnector
%QBA4	Remote controlled load breaker
Insulators	**Type**
%UAA1	Sectioning overlap
%UAA2	Section insulator

The type aspect can also provide context to technical systems. In this project, defining types for different station configurations was useful. The types defined in table 3.2 describe four different station configurations. These are helpful for the algorithm as they help draw parallel station lines correctly location-wise.

Table 3.2: Different technical system types

System for control of electrical energy flow	Type
%KL1	Regular station with parallel line under
%KL2	Regular station with parallel line over
%KL3	Regular station with the parallel lines over and under

3.3.9 Edge classification in the connection graph

If the source component in a connection row is not already drawn, the algorithm must figure out how to draw both the source and target components without knowing their relative location. This version of the algorithm assumes that the connection graph has *one* root node, which is an ancestor for all other nodes in the graph. After the graph is topologically sorted, the first connection's source node is always the root node since all other nodes depend on it. The first connection can then be treated as a special case, and the algorithm can draw this connection somewhere desirable. Since all other connections indirectly depend on the root node, all other connections can be drawn normally.

The closer the graph is to a tree, the easier it is to draw components accurately. A tree structure is trivial to draw. However, the task becomes exponentially more complicated if the graph contains many cross edges. Back edges, which create cycles, are unmanageable for the application as topological sorting algorithms do not work on cyclic graphs. Forward edges generally do not disturb the tree-like structure too much, and they should not be common occurrences as they indicate direct connections between two components over long distances. Cross edges, on the other hand, have reasons to exist and do cause complications for the algorithm. Cross edges connect two branches of the DAG, as shown in figure 3.16. Cross edges do not cause any issues for the topological sorting algorithm, as they do not cause cycles. This means the drawing algorithm still always has a valid sequence of connections to draw. However, cross edges often indicate complex branching systems that may be difficult to draw with regard to the system it reconnects with. In other words, cross edges frequently introduce relative location issues. For instance, consider the example graph in figure 3.16. Since the algorithm uses a topologically sorted list of connections, $(1, 2)$, $(1, 3)$, $(2, 4)$, and $(2, 5)$ will be the first connections considered. After the algorithm processes them, two edges remain: $(3, 6)$ and $(5, 6)$. Since they are identical dependency-wise, which edge comes first is an arbitrary choice for the topological sorting algorithm. So unless the algorithm develops a way to consider multiple edges concurrently, node 6 will be drawn with respect to either node 3 or 5. Consequently, the other connection will be processed, and it is unclear what the algorithm should do. This is precisely why the ideal graph structure is when all nodes have exactly *one* direct dependency. Although cross edges are not tackled in this project's application, possible workarounds and solutions will be discussed in a subsequent section.

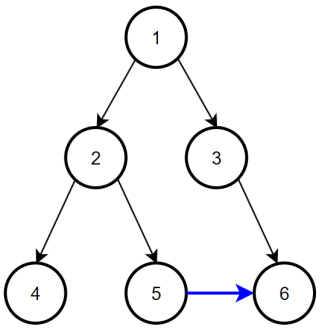


Figure 3.16: A DAG where the blue arrow indicates a cross edge between nodes 5 and 6

3.3.10 Connection graph manipulation

It was discovered that it was possible to represent relatively simple connection graphs as trees. However, trees come in varying shapes and forms; some were considered more appropriate. Height-balanced trees, as defined in section 2.1.2, and close to height-balanced trees, are difficult for the algorithm to draw accurately. From the root's perspective, its descendants branch off into several sizable subtrees, making it more challenging to accurately draw subsystems relative to each other. However, if the tree is more unbalanced, the branching subtrees will be shorter and easier for the algorithm to handle. The optimum is an unbalanced tree containing a distinct *spine*. In this project's scope, a spine refers to the edges and nodes in the connection graph that comprise the main structure. It can also be defined as the longest path from the root node to a leaf in the graph if the graph is a tree. If the DAG is a tree, the nodes of the spine are internal nodes of the tree, while branching subsystems end up as leaves. Identifying a spine's existence and what nodes and edges it consists of is challenging, but sufficient modeling rules can make the occurrence of a spine predictable. To be clear, a spine is not a widely used term in graph theory but is merely used to clarify this project's most desirable graph structure.

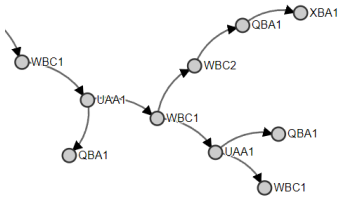


Figure 3.17: An example of how the connection graph works as a tree structure

Figure 3.17 shows a typical structure in subtrees of the connection graph. The `WBC1`, `UAA1`, `WBC1`, `UAA1`, and `WBC1` components compose the spine of the tree and are internal nodes (except for `WBC1` as this is the last component in the spine). Branching subtrees are of limited size and do not cross back into the spine, which would indicate the existence of unwanted cross edges. `WBC2` is a parallel station line. Even though the parallel line can be considered connected to `WBC1` and both `UAA1`'s, it also makes sense to regard it as a subsystem that branches off the main line. If this consideration is applied as a modeling rule, the stations do not break any tree properties in the graph. In the figure, `WBC2` is the root of a subtree to which a switch and transformer component is connected, where the transformer is a leaf. These types of design choices are crucial to upholding the algorithm's consistency.

3.3.11 Additional API endpoints

The concepts explored in the previous sections form the foundation of the final algorithm. However, it was beneficial to go further and design additional utility API endpoints. These endpoints serve to enhance the efficiency of front-end development. For instance, since the connections only contain the corresponding object IDs, retrieving other component attributes is often necessary when only IDs are available. To address this, API endpoints were designed to create maps, mapping object IDs to other object attributes. One endpoint provides mappings between object IDs and object RDs, while another maps object IDs to object types. This approach empowers clients to dynamically fetch these maps, ensuring they always have access to the most up-to-date mappings, thereby streamlining their development process.

The handler function implementations use Go hash maps and populate them by looping over the fetched objects from the database. These maps map between integers and strings, as object IDs are integers, and types and RDs are strings. The maps are then returned to the client as a JSON-encoded map.

3.3.12 Interactive RDS visualization

Additional design ideas emerged as the drawing algorithm began reaching its final form. The algorithm worked well with compatible data input. Still, as this project is all about the application of RDS, it seemed interesting to develop a way to visualize the system's RDS structure in the software. A way to visualize this is to draw boxes with single-level RDs around the systems, as in section 3.1.2. The drawn components data structure contains relevant information about where each component object is drawn on the canvas. Drawing boxes around drawn component systems is possible, but drawing their single-level RDs as text next to them was considered sufficient. Drawing perimeters around component systems would lead to unnecessary complexity on the canvas. However, technical and power supply systems can be visualized by drawing perimeters around their encapsulated systems.

This project adheres to the separation of concerns design principle. This principle is beneficial because it separates the component drawing loop from the implementation of the RDS box drawing. The drawing algorithm for drawing boxes and RDs is only initiated when the component drawing algorithm is finished. This ensures that the drawn components are all appropriately initialized.

The application presently consists of two main algorithms: the component drawing algorithm and the box drawing algorithm, each focusing on specific concerns.

The box drawing algorithm loops through the drawn components list, making decisions based on the context. The algorithm tracks the technical and power supply systems of which the drawn components are part. The outline of the technical system drawing algorithm is as follows: a *current technical system* variable is initialized, containing the single-level RDs of the first technical system in the drawn components array. The coordinates of the first component system in the array are also initialized to variables before the loop starts. Whenever the algorithm enters a component system that belongs to a technical system that does not match the current technical system variable, the algorithm draws a box corresponding with the current technical system. Afterward, the current technical system variable is set to the new one. In the same iteration, the coordinates are replaced by the current iteration's component system coordinates. The boxes are drawn using the coordinate variables representing the first components in the corresponding technical systems. Since the boxes should form perimeters around all relevant components, the y-coordinate must be offset vertically so that the perimeters do not interfere with the drawn components. This offset is a constant that ensures enough space for all possible systems. The rectangle's width is then found by subtracting the first component's x-coordinate from the last component's x-coordinate. The box height is also a hard-coded constant. Some minor numerical tweaks are done in the code so the boxes do not intersect.

4 Results

4.1 RDS models

The following subsections present visualizations of how the different model alternatives were implemented. All models model a system bounded by Lundamo converter station and Stavne.

4.1.1 Alternative 1

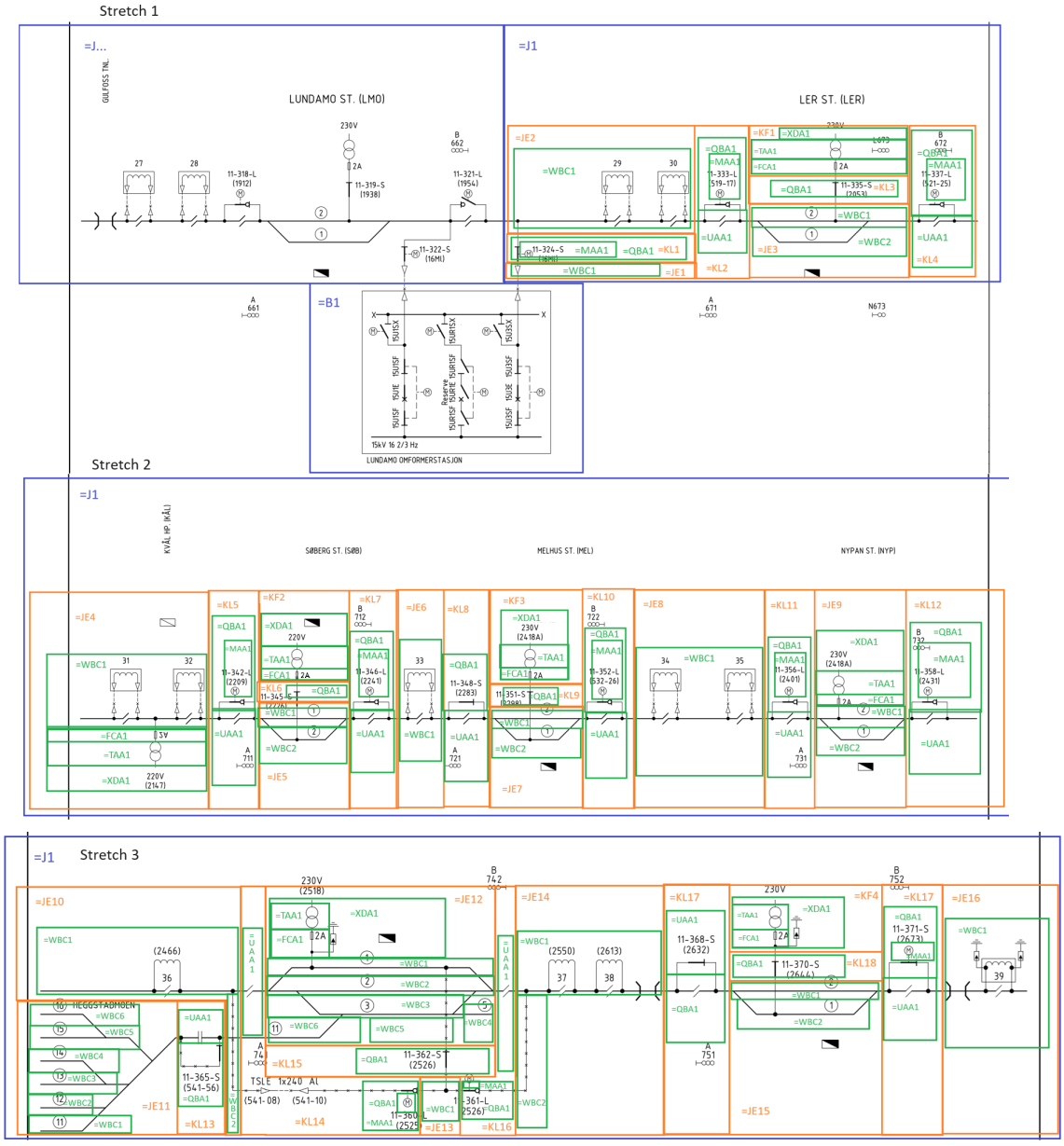


Figure 4.1: Implementation of alternative 1 on a stretch between Lundamo and Stavne

4.1.2 Alternative 2

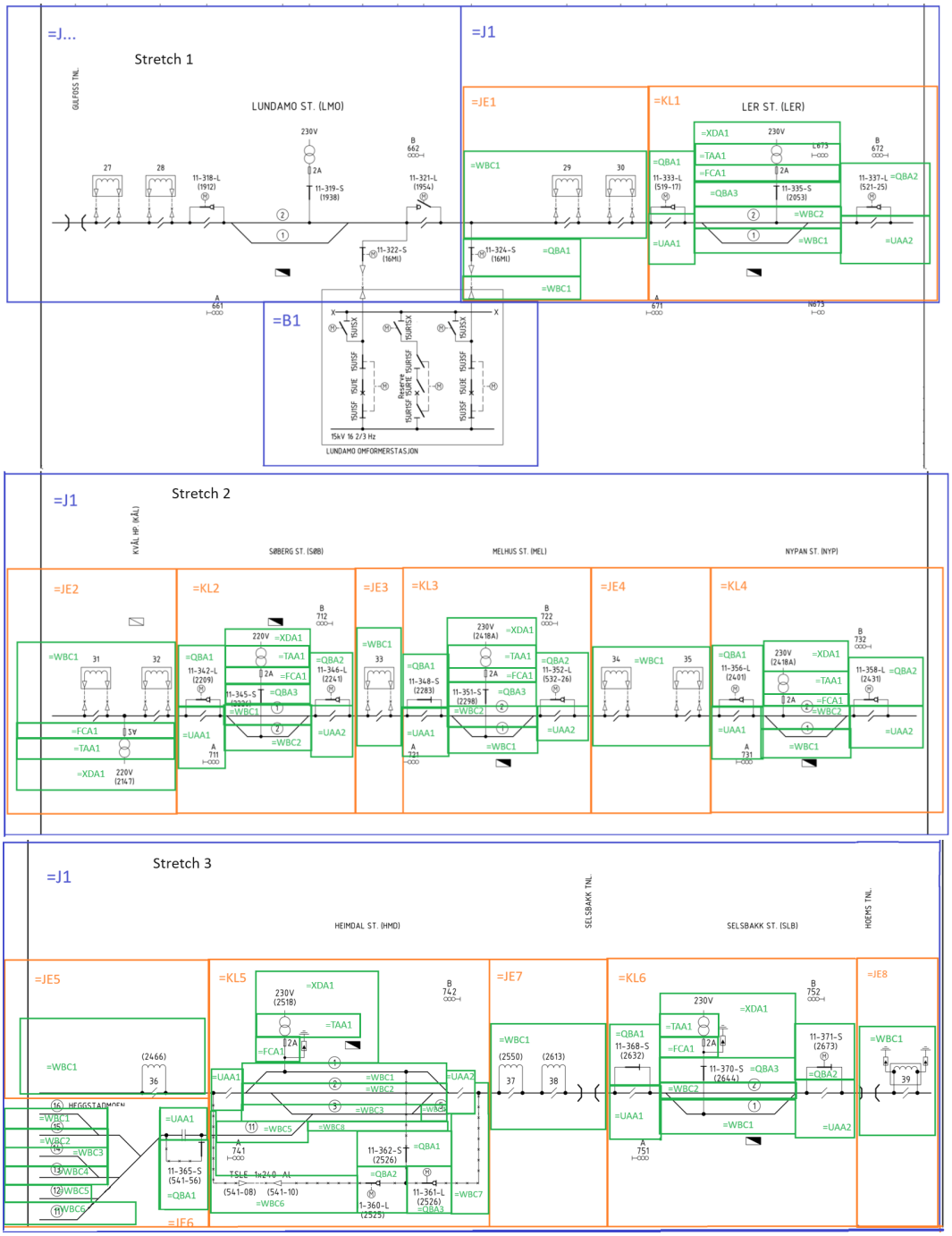


Figure 4.2: Implementation of alternative 2 on a stretch between Lundamo and Stavne

4.1.3 Alternative 3

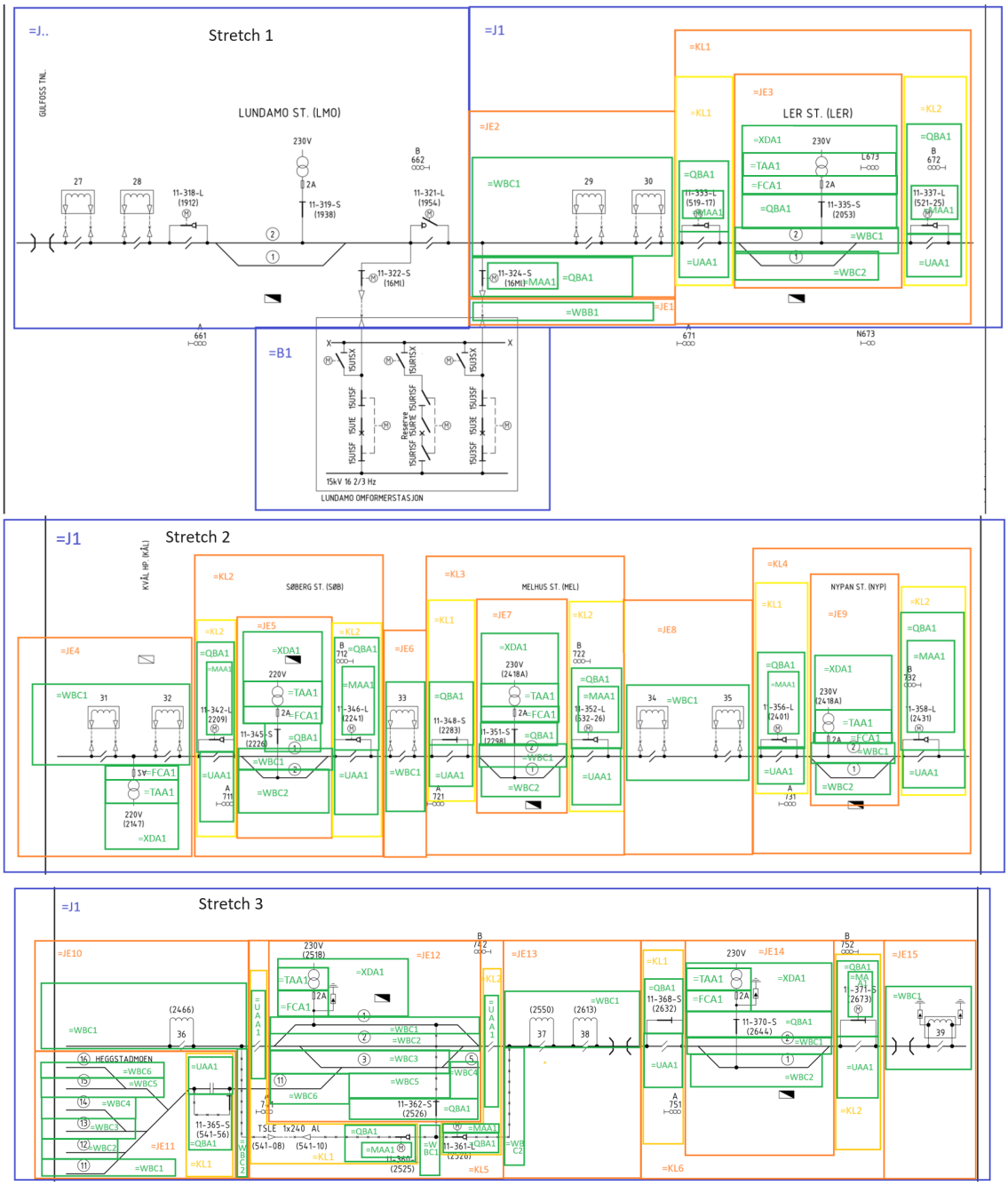


Figure 4.3: Implementation of alternative 3 on a stretch between Lundamo and Stavne

4.1.4 Alternative 4

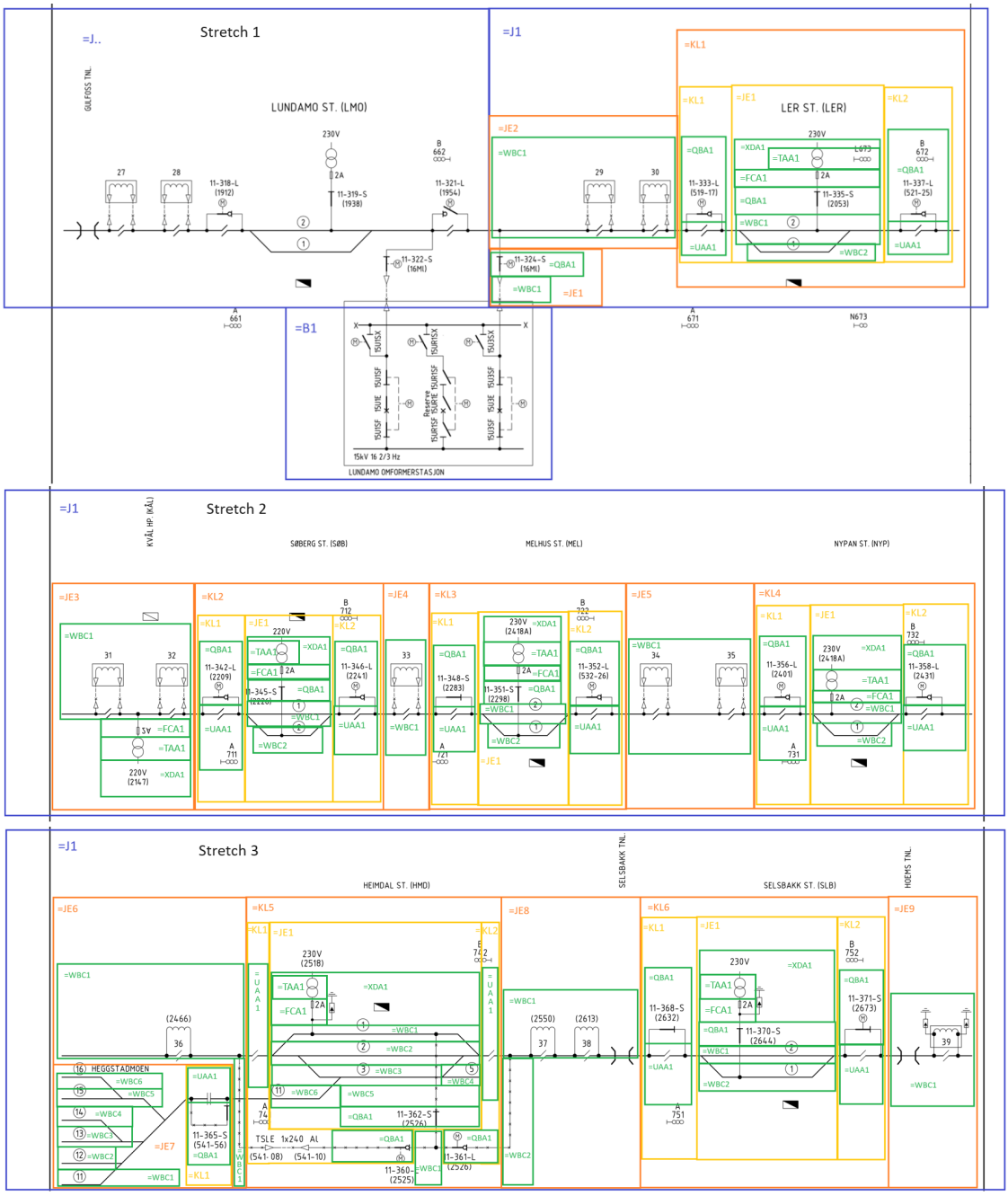


Figure 4.4: Implementation of alternative 4 on a stretch between Lundamo and Stavne

4.2 RDS trees

The subsequent trees illustrate RDS model structures.

4.2.1 Alternative 1

There are 39 technical systems, 40 including the one in B. For readability, the tree is split into two figures: figure 4.5 and figure 4.6.

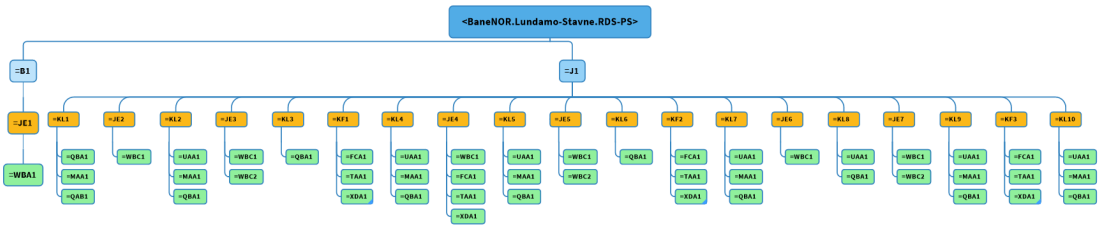


Figure 4.5: Tree of alternative 1, part 1

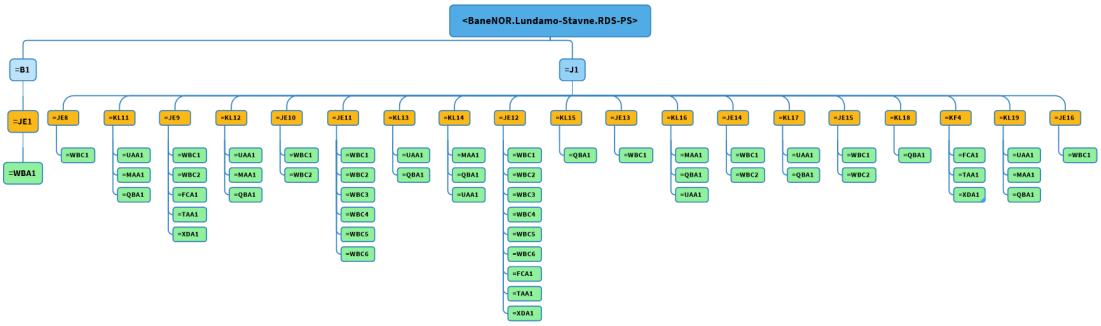


Figure 4.6: Tree of alternative 1, part 2

4.2.2 Alternative 2

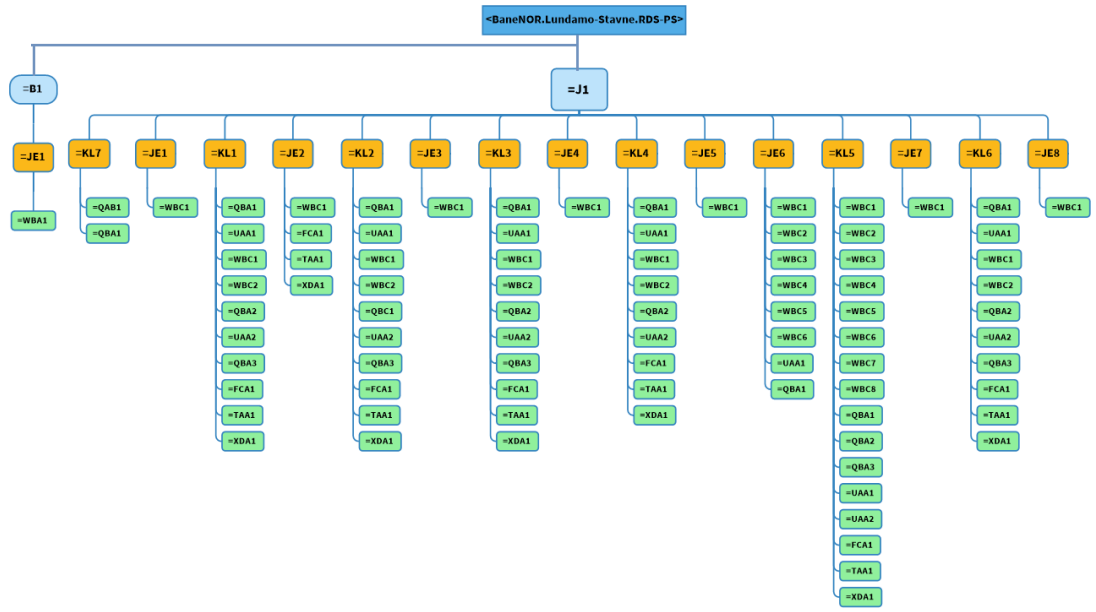


Figure 4.7: Tree of alternative 2

4.2.3 Alternative 3

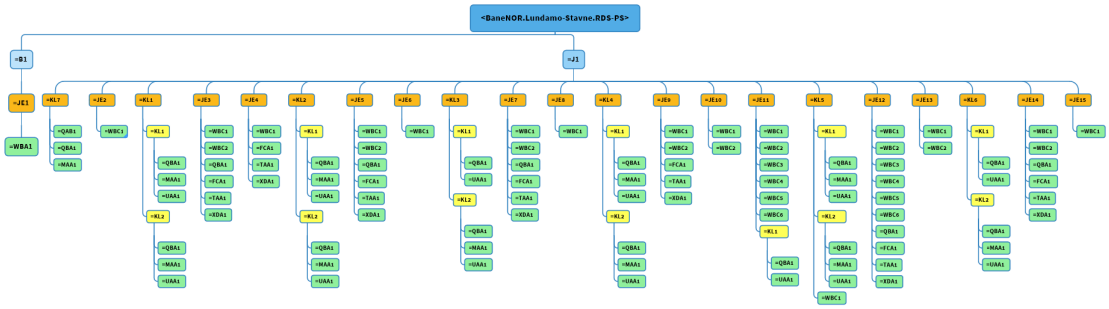


Figure 4.8: Tree of alternative 3

4.2.4 Alternative 4

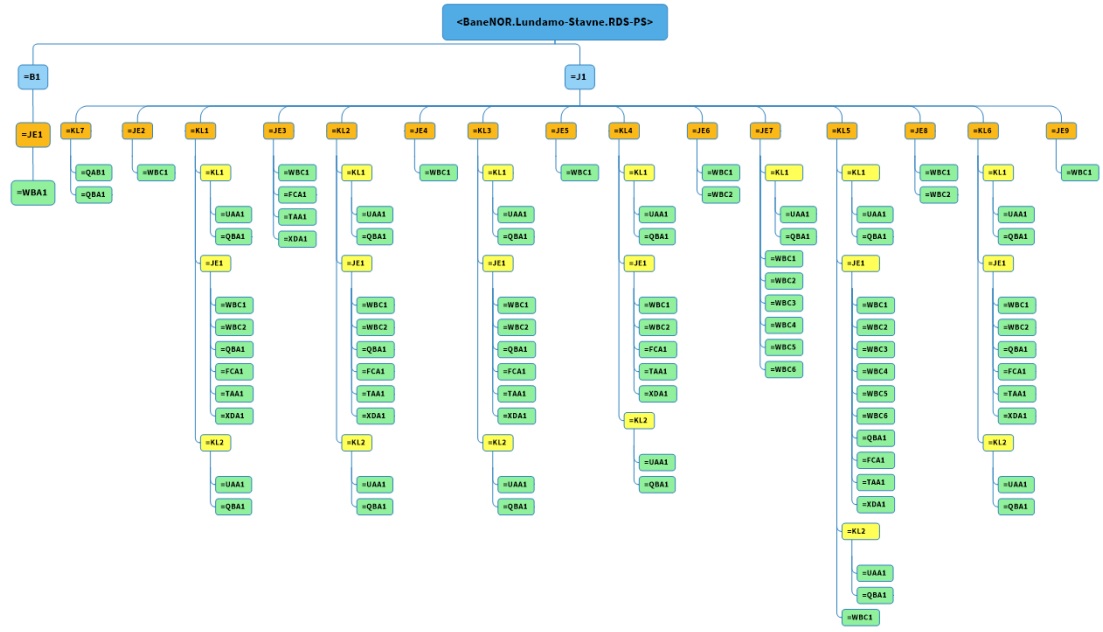


Figure 4.9: Tree of alternative 4

4.3 Clarifications

In alternatives 1 and 3, motors are modeled as separate component systems. This is because the type aspect was not yet implemented when these alternatives were developed. When alternative 2 was in development, motors were not considered an important part of the system. They were, therefore, not modeled. Later, when the type aspect was introduced, alternative 4 implemented motors in combination with switches as a type.

A discrepancy between the implementations and the trees is the line between B and J on the first stretch. Firstly, this line is implemented differently in every single implementation, meaning a rule should be established for what to do in this scenario. It was decided that all these line should be swapped with a KL, including the disconnector and the circuit breaker inside of B, because of the receiver ownership principle.

The B systems have a technical subsystem, which is included in the trees to show how the J systems are connected to the busbar in the B systems. However, these technical systems are not modeled in the implementation.

4.4 Types

In this project, types were developed for component, technicals systems. Table 4.1 shows the component system types, table 4.2 shows types for technical systems.

Table 4.1: Different component types

Switches	Type
%QBA1	Disconnector
%QBA2	Load breaker
%QBA3	Remote controlled disconnector
%QBA4	Remote controlled load breaker
Insulators	**Type**
%UAA1	Sectioning overlap
%UAA2	Section insulator

Table 4.2: Different technical types

System for control of electrical energy flow	Type
%KL1	Regular station with parallel line under
%KL2	Regular station with parallel line over
%KL3	Regular station with the parallel lines over and under

Figure 4.10 shows the different station variations `%KL1`, `%KL2`, and `%KL3` describe.

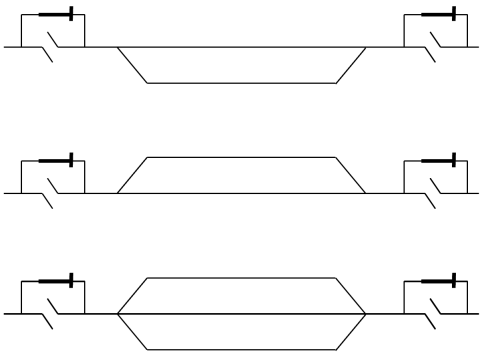


Figure 4.10: Illustration of the different station types. From top to bottom %KL1, %KL2, %KL3

4.4.1 Alternative 4 with the type aspect

Figure 4.11 shows a collapsed version of figure 4.9 in addition to the applied types.

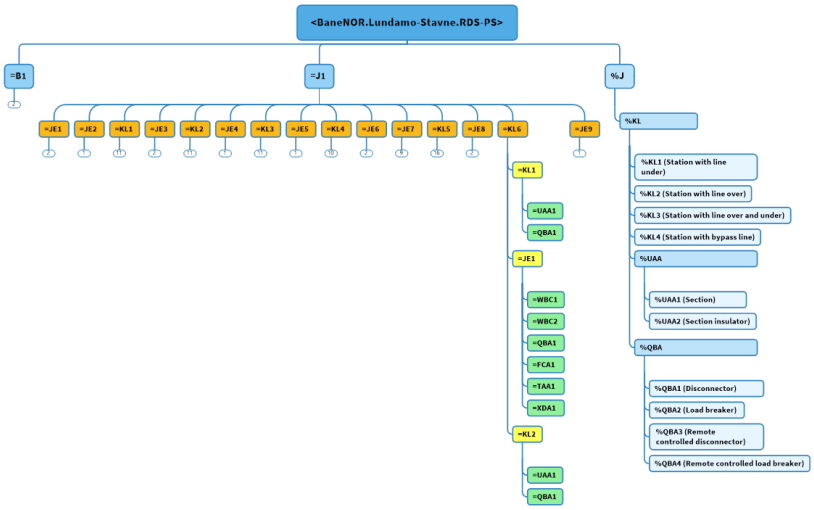


Figure 4.11: RDS tree displaying the function and type aspect with all technical systems except KL6 collapsed

Furthermore, links are added between types and objects, as shown in figure 4.12.

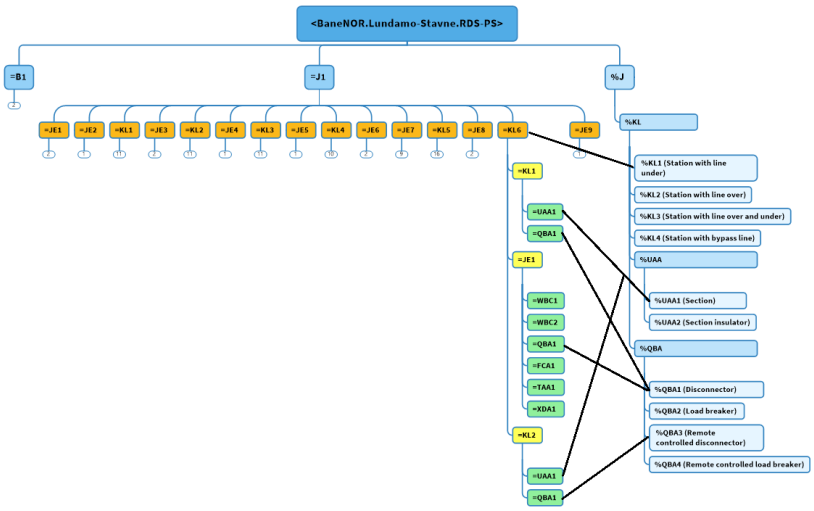


Figure 4.12: Type aspect implementation, only considering =J1.KL6

All object RDs from figure 4.12 can be seen in table 4.3.

Table 4.3: RDS tags from tree in figure 4.12

System	Functional aspect tag	Type aspect
Main system		
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1	
Level one technical system		
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6	%KL1
Level two technical system		
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL2	
Component system		
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL1.UAA1	%UAA1
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL1.QBA1	%QBA1
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.WBC1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.WBC2	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.QBA1	%QBA1
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.FCA1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.TAA1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.JE1.XDA1	
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL2.UAA1	%UAA1
	<BaneNOR.Lundamo-Stavne.RDS-PS> =J1.KL6.KL2.QBA1	%QBA3

The RDs in table 4.3 are the results of links between the functional- and type aspects. All RDs with applied types are available in the appendix.

4.5 Database

Figure 4.13 shows a table segment describing the objects' RDs, their types, and IDs.

rd ltree	id [PK] integer	type text
baneNorLundamoStavne.J1.JE1.WBC1	1	[null]
baneNorLundamoStavne.J1.JE1.QBA1	2	[null]
baneNorLundamoStavne.J1.JE2.WBC1	3	WBC1
baneNorLundamoStavne.J1.KL1.KL1.UAA1	4	UAA1
baneNorLundamoStavne.J1.KL1.KL1.QBA1	5	QBA4

Figure 4.13: Segment of the object table

The type column is linked to the type table, shown in figure 4.14, by a foreign key.

value text	id [PK] integer
WBC1	1
WBC2	2
WBC3	3
WBC4	4

Figure 4.14: Segment of the type table

The ID column from the object table explains relationships between objects further. These relationships are of the type "is connected to." Figure 4.15 shows a segment of the connection table. The `object1` and `object2` columns are IDs taken from the object table and used as foreign keys. The relationships of foreign keys are shown in figure 3.10.

id [PK] integer	object1 integer	object2 integer
3	3	4
4	4	5
5	4	6
6	6	7
7	6	8

Figure 4.15: Segment of the connection table

4.5.1 Clarifications

As seen in figure 4.13, there is no <> to denote the top node. This is due to the limitation of the `ltree` data type. Also, the top node does not provide information about what part of RDS is used, contradicting the guidelines given in 2.3.7. This mistake was spotted towards the end of the project and was therefore not corrected. The whole database is given in appendix B.

4.6 Software

4.6.1 API

The API played a crucial part in the project, bridging the gap between the front-end and back-end. Even though Python was initially chosen to create the API, it was discarded in favor of a faster and more efficient Go API. The API consists of five endpoints: `objects`, `connections`, `idrd`, `idtype`, and `query`. The object and connection endpoints provide data from their respective PostgreSQL database tables, while `idrd` and `idtype` supply maps from component ID to RD and type, respectively. The `query` endpoint moves a specified subtree to a specific location in the underlying `ltree` implementation. The connections handler function incorporates a topological sorting algorithm defined inside the API. All features are accessible in appendix A.

4.6.2 Front-end

The front end receives model data from the API and generates single-line diagrams. The front-end results are split into three separate but connected modules: data structures, algorithms, and generated single-line diagrams.

The two algorithms developed in this project both use the JavaScript class `ComponentState`. This data structure holds relevant information about drawn components, as visualized in figure π. Instantiated `ComponentState` objects are applied in the algorithms through the `drawnComponents` array. This array contains all important information about components on the canvas.

Maps fetched from the Go API are stored in JavaScript `Maps`, which work similarly to Go maps. Data received from the `objects` and `connections` endpoints are kept in variables annotated with apt types in JSDoc. All custom JSDoc and other types utilized in the project are available in appendix A.

Figure 4.16 shows the diagram generated when the entire application is run on the electrical power supply system from Lundamo converter station to Stavne. The figure is the result of the single-line diagram algorithm executed in combination with the RDS box drawing algorithm.

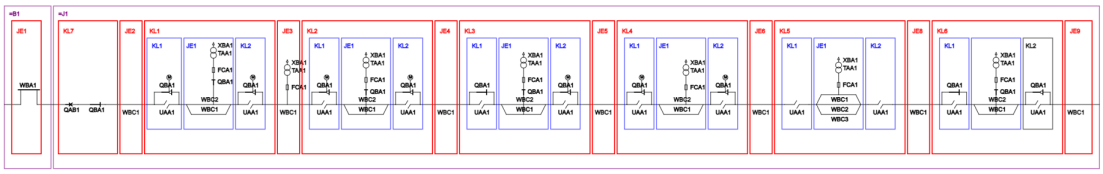


Figure 4.16: Automatically generated single-line diagram for Lundamo-Stavne

Figure 4.16 shows the program applied on a relatively large system. On the other hand, figure 4.17 shows a smaller system with slightly tweaked system configurations. This system displays the many types the algorithm can handle. For instance, `=J1.KL2.UAA1` is a section insulator, which has the same class code as a sectioning overlap but is distinguished visually by the type aspect. The single-line diagram also implements the three different KL types.

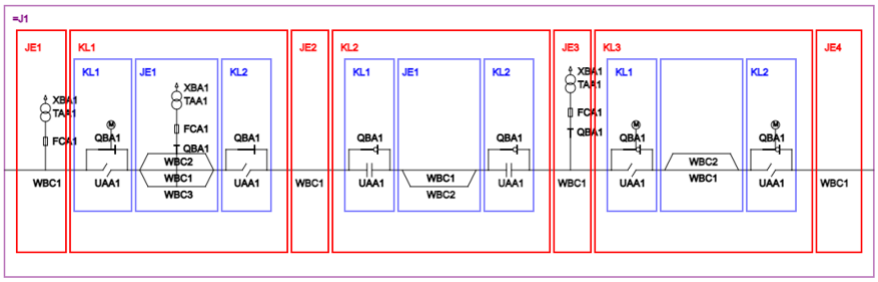


Figure 4.17: Automatically generated single-line diagram for a custom-made small system

In the software released with this thesis, the box drawing algorithm can be switched on and off by the user with the click of a button. Figure 4.18 displays how the single-line diagram looks when the client disables the box drawing algorithm.

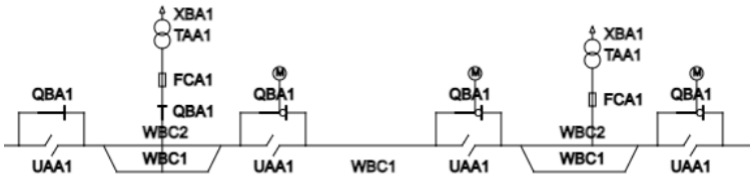


Figure 4.18: Automatically generated single-line diagram for a two-station stretch in Lundamo-Stavne with box drawing algorithm disabled

5 Discussion

5.1 Alternate RDS implementations

This project explores four ways to implement RDS for railway power supply systems. The implementation of the alternatives can be seen in the results in section 4.1, with the RDS trees in section 4.2. This section will discuss their strengths and weaknesses.

5.1.1 Strength and weaknesses of overlapping and edge-to-edge

Edge-to-edge was chosen instead of an overlapping configuration because it is easier to utilize the data when every component system is associated with only one technical system. As expected, the data gathered using the edge-to-edge implementation style was relatively easy to utilize. Each component only has one multi-level RD, which benefits most practical use cases, including algorithm development.

Even though an overlapping configuration was not used in this project, it does result in fewer technical systems, which is a desirable feature. However, component system objects that are part of linking systems will get multiple multi-level RDs, which causes ambiguity at the component system level.

5.1.2 Strength and weaknesses of the alternatives

Implementation of alternative 1 can be seen in figure 4.1, and the resulting tree can be seen in figure 4.5 and 4.6. The `J1` power supply system has 39 technical systems as its children, which is too many according to section 2.3.6.2. Even though the system modeled in this project is relatively large, for smaller systems the resulting structures are detailed and easy to understand. Alternative 1 is also relatively simple to implement.

Implementation of alternative 2 can be seen in figure 4.2, and the resulting tree can be seen in figure 4.7. As seen in figure 4.7, there are fewer technical systems than alternative 1. However, this causes technical systems to have more component systems each. For instance, `=KL5` has 16 sub objects. This is within the rules described in section 2.3.6.2. However, there were also some negative experiences regarding this implementation. When many component systems are part of the same technical system, it becomes difficult to understand the system structure. Alternative 2 is too simplistic because its technical systems provide insufficient detail.

Implementation of alternative 3 can be seen in figure 4.3, and the resulting tree can be seen in figure 4.8. The resulting RDS tree is narrower than alternative 1. Subsidiary technical systems cause technical systems to have fewer component systems each. Consequently, the stations are more thoroughly described. However, knowing whether a `JE` system is associated with a station is difficult.

Implementation of alternative 4 can be seen in figure 4.4, and the resulting tree can be seen in figure 4.9. Alternative 4 has the same amount of main technical systems as alternative 2, but details the stations to a greater extent. Alternative 4 makes `JE` systems associated with the station subsidiary systems to the main `KL` system.

5.1.3 Overall implementation thoughts

The preceding discussion indicates that alternative 4 is superior; however, alternative 1 could be deemed more fit for smaller systems, and with few component systems, alternative 2 can be a good fit. Alternative 4 is an improved version of alternative 3.

This project is limited to a small subset of system configurations that exist in the railway power

supply system. Consequently, the rules and principles developed in this project are not necessarily appropriate for all possible system configurations.

For instance, *bypass lines* seen in KL5 in alternative 4 in figure 4.4. The rules established for this project do not definitively tell what to do when these are encountered. This means that the work ahead should include establishing some rules regarding bypass lines to make implementations unambiguous.

As mentioned in 4.3, this project models the connection between B and J systems as a simple bus bar with a circuit breaker and a disconnector. However, rules for describing B systems to a greater extent have yet to be developed throughout this project. Such a description could lead to greater accuracy.

5.2 Database discussion

5.2.1 Structure

A relational database enables the user to store information in a structured manner. Foreign keys make connecting tables and referencing rows from other tables possible. The connection table references the object table, representing a many-to-many relation between objects. The object table also includes a foreign key constraint to the type table. The object-type relation can be represented as a single column as it is a one-to-many relation where each object can only have one type.

Optionally, the database could be structured fully based on relationships. In such a structure, one would construct a table of *relationship types* with relationship types such as "is connected to", "is child of", and "is of type". Another table would list all system object single-level RDs. A separate table would then consist of three columns: two references to single-level RDs and one to the relationship type table. This is a reasonable approach, but `ltree` essentially provides a built-in implementation of the "is child of" relationship type, making development easier. Since "is of type" is a one-to-many relation, it can be represented as a single column in an object table, simplifying the database structure. Additionally, if a client wants to find the type of an object, it is much faster to check its type attribute than to query a relationship table. However, if the model needed more many-to-many relations, the relationship type structure would become more suitable.

Although `ltree` seems like an ideal representation of RDS tags, moving branches around in the underlying tree structure is not trivial. In the relationship type model, multi-level RDs are implemented as sequences of "is child of" relations. For instance, only one row must be modified to move an entire technical system subtree to another power supply system. To perform a similar action with the `ltree` implementation, one must either change every label path in the subtree or implement a custom function for this purpose. This project implements an API endpoint that moves a subtree to a specified location.

5.2.2 ltree

`ltree's` structure matches the multi-level RD structure, making it a good choice for RD representation. Functions and operators within the extension allow for deleting, moving, and copying branches, making it possible to update the hierarchical tree structure to match the physical specifications.

5.3 API

The API developed in this project connects the database to the front end. The front end can quickly and dynamically fetch all relevant information in the database via the API. In addition, Go is considered a swift programming language, giving clients smooth experiences when interacting with the API.

Web APIs, such as the one created in this project, are relatively streamlined in how they operate. Additionally, as Go has all the necessary API tools built into the standard library, there was little need for ingenuity when constructing them. This fact, in combination with familiarity with the Go programming language, led to a smooth development process in which the focus could be directly on what functionality the client side required. As a consequence, minimal time was spent on actual API programming. Instead, emphasis was placed on structuring the API and brainstorming useful API functionality.

The API's structure follows the separation of concerns principle, as described in section 2.7.1. Each API endpoint is designed to deliver data related to specific concerns. For instance, if the client needs a map that maps between object IDs and object types, the client can send a request to the endpoint designated to this exact concern. This API structure ensures that the bare minimum of data is sent on each response; only the relevant information is sent to the client. A reasonable argument against this approach is that the client might need to send multiple requests to obtain the required data. However, if numerous independent data structures are returned together from an endpoint, it may be difficult for the client to parse the response correctly. As each endpoint handler function intends to fulfill a specific purpose, the developer will have a clear mental image of what the function should do. Accordingly, separation of concerns often causes easier development, performance improvements, and clarity for the consumer.

5.4 Front-end

The front-end of this project encompasses all functions, algorithms, and data structures related to canvas drawing. Developing the algorithm proved to be a creative process, requiring constant adaptation to changes in the problem space. Throughout the project's course, the algorithm underwent substantial changes to meet unforeseen requirements. Key challenges included addressing issues related to location accuracy, generality, complexity, and graph structure.

5.4.1 Location accuracy

Spatial accuracy posed a persistent challenge due to the unavailability of real location data, making it impossible to achieve complete accuracy in drawing components. The primary goal of the application, however, is to visualize the system's structure in terms of its power transmission function. Thus, exact spatial accuracy is unnecessary if components are correctly positioned relative to each other to depict the system's functionality. Introducing the connection table allowed for relative positioning between components, although it required analyzing one relation at a time. This limitation made it difficult to accurately position technical and power supply systems relative to each other. To mitigate this, the algorithm maximized the use of contextual information, introducing the concept of types to provide necessary context.

5.4.2 Types and contextual information

Types made RDS objects more descriptive, enabling the algorithm to make more informed decisions. For instance, if parallel station line objects have associated types, the algorithm can correctly position them relative to the main line. These types are necessary for the algorithm to avoid ambiguous situations. Another way to provide more context is to consider the different parts of the multi-level RDs. Component system objects are often configured differently depending on the technical and power supply systems they are part of. For instance, as of alternative 4, the algorithm can recognize when it is considering objects that are part of a station. KL systems enclose stations, and the station lines are encased by JE systems inside the KL systems. Since this configuration is specific to stations, the algorithm can use the context to draw accordingly. The usefulness of the context depends on the way the model is structured. As mentioned, alternative 4 makes it possible to identify stations unambiguously, which is not the case for the other alternatives.

5.4.3 Generality

Generality is crucial for algorithm applicability across various client systems. An algorithm tailored to a specific system may become ineffective for others if it relies on customized logic. Although the algorithm developed in this project is not universal, following the modeling principles proposed in this thesis can yield similar results for other systems. The type aspect exemplifies this light form of generality, as the algorithm depends on properly applied types to draw components correctly.

5.4.4 Complexity

Increasing complexity demands richer contextual information. While the systems used in this project are relatively simple, more complex systems result in larger decision trees, potentially leading to a disordered program. To manage complexity, the primary logic was designed to be as universal as possible, with custom functions handling specific complexities at the decision tree leaves. Well-documented and descriptive custom functions help maintain orderliness.

5.4.5 Structural complexities

Structural complexities arise from deviations from an optimal relation graph tree structure, often requiring significant changes in the main algorithm logic. Complex systems make it harder to model connections as a tree, with subsystems branching off and reconnecting further down the graph. Cross edges, connecting two branches, present a challenge but can be managed with minor process tweaks. However, a graph with multiple source nodes necessitates a complete algorithm overhaul. Running the algorithm separately for each source node could be a potential solution, though it reintroduces location issues. Tracking coordinates to choose unoccupied locations on the canvas for new spines may help, but without additional location data, accurate relative positioning remains unachievable. Consequently, drawing single-line diagrams for connection graphs with multiple source nodes was deemed too comprehensive for this project.

6 Sustainability goals

This project makes contributions to the United Nations' sustainability goals 9 and 13. Goal 9 emphasizes the importance of building resilient infrastructure, promoting inclusive and sustainable industrialization, and fostering innovation. By enhancing the efficiency, reliability, and standardization of electrical infrastructure in the railway sector, this project supports these objectives, ensuring that railway systems are robust, innovative, and capable of meeting future demands.

In addition, the project supports Goal 13, which calls for urgent action to combat climate change and its impacts. By facilitating the electrification of railway networks, the project helps to reduce dependence on fossil fuels and lower greenhouse gas emissions. This transition to electric railways is a crucial step in mitigating climate change and reducing the environmental footprint of transportation.

BaneNOR's efforts to develop a fully electrified railway system are in perfect alignment with these sustainability goals. The project not only advances the technical and operational aspects of railway infrastructure but also promotes a sustainable and environmentally friendly transportation system. This dual focus on innovation and sustainability underscores the project's broader impact on promoting a greener and more resilient future.

7 Road Ahead

The development of an automated system for generating railway single-line diagrams using RDS in accordance with ISO/IEC 81346 standards marks the beginning of a broader journey. This project serves as a proof of concept where several areas for future research and improvement have been identified.

- **Scalability and Performance Optimization**: Optimization is necessary to handle larger datasets and more complex railway networks. Future research should focus on improving the scalability and performance of the software to accommodate further demand.
- **Integration with Existing Systems**: Further work is needed to integrate the automated single-line diagram generation system with existing railway power supply management and control systems. This integration ensures seamless data flow and enhances the overall efficiency of railway power supply system operations. This includes establishing concrete rules regarding bypass lines and B systems.
- **User Interface and Usability**: Developing a more user-friendly interface will be crucial for the widespread adoption of the software. Enhancing the usability will enable clients to interact with it more effectively, improving its practical utility.
- **Enhanced Algorithm Development**: Continued refinement and enhancement of the algorithms for generating single-line diagrams, such as those for topological sorting and component state tracking, will be essential. Exploring advanced machine learning techniques and using location data could further improve the accuracy and efficiency of the system.
- **Standardization and Compliance**: As standards evolve, keeping the system updated with the latest RDS and other relevant standards will be important.
- **Extension to Other Industries**: While this thesis focuses on railway power supply systems, the principles and methodologies developed here can be adapted for use in other industries that require complex system documentation, such as aerospace, automotive, and energy sectors. Exploring these applications can open new avenues for research and development.
- **Real-world Testing and Feedback**: Implementing the system in real-world railway projects and gathering feedback from industry professionals will provide valuable insights for further refinement. This iterative process will help identify potential improvements and ensure the system meets the practical needs of its clients.

The road ahead is filled with exciting opportunities for further research, development, and application of the automated single-line diagram generation system. Addressing these challenges and exploring new possibilities will advance the field of railway power supply documentation and contribute to the efficiency of railway operations worldwide.

8 Conclusion

This thesis has demonstrated a systematic approach to automating the generation of single-line diagrams for railway systems using the RDS. The project encompassed three main phases: developing various modeling alternatives, establishing a relational database, and creating an algorithm for single-line diagram generation.

Initially, four different modeling alternatives were considered, with the chosen alternative utilizing a station-oriented structure with subsidiary technical systems. This provided a robust framework for representing the railway's electrotechnical aspects.

Subsequently, a relational database was established, ensuring efficient data storage and management through the use of primary and foreign keys, SQL, and the ltree extension. This database served as the foundation for the final phase of the project.

The final phase involved developing an algorithm capable of generating single-line diagrams automatically. This algorithm incorporated techniques such as topological sorting, component state tracking, and decision trees, successfully creating reliable single-line diagrams.

Despite these achievements, the project has identified several limitations and areas for future research. The current model is specifically tailored to the railway stretch between Lundamo and Stavne, requiring adjustments for broader applicability. Additionally, ongoing developments in the RDS standard may necessitate future modifications to both the modeling and algorithms.

Future research should focus on scalability, performance optimization, integration with existing systems, enhancing user interfaces, and refining algorithms. Implementing the system in real-world projects and obtaining industry feedback will be crucial for further refinement and practical application.

In conclusion, this thesis has successfully validated the concept of automated single-line diagram generation, marking a significant step forward in standardizing railway documentation and improving operational efficiency in the railway industry.

References

[1] K. Thulasiraman and M. Swamy, "Graphs: Theory and algorithms," in John Wiley and Sons, 1992, p. 118.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2022.

[3] Jernbanekompetanse, "Elkraft/elektrisk kraft i 100 år," Apr. 2024, Accessed on May 19, 2024. [Online]. Available: https://jernbanekompetanse.no/wiki/Elkraft/Elektrisk_kraft_i_100_%C3%A5r.

[4] E. Csanyi, "Differences between disconnectors load switches, switch disconnectors and circuit breakers," Jun. 2014. [Online]. Available: https://electrical-engineering-portal.com/disconnectors-load-switches-switch-disconnectors-cbs#3.

[5] T. E. of Encyclopædia Britannica, "Transformer," Mar. 2024, Accessed on April 18, 2024. [Online]. Available: https://www.britannica.com/technology/transformer-electronics.

[6] Circuitglobe, "Booster transformer," 2018, Accessed on April 18, 2024. [Online]. Available: https://circuitglobe.com/booster-transformer.html.

[7] Jernbanekompetanse, "Definisjon:sugetransformator," Sep. 2016, Accessed on May 19, 2024. [Online]. Available: https://trv.banenor.no/wiki/Definisjon:Sugetransformator.

[8] RS, "A complete guide to fuses," Jan. 2023, Accessed on April 22, 2024. [Online]. Available: https://uk.rs-online.com/web/content/discovery/ideas-and-advice/fuses-guide.

[9] "Industrial systems, installations and equipment and industrial products - Structuring principles and reference designations - Part 1: Basic rules," IEC 81346-1, Standard, Mar. 2022.

[10] *Rds 81346 for power systems*, Accessed: 2024-05-15, 2022. [Online]. Available: https://static1.squarespace.com/static/621c9f78c59da237abfc2b84/t/62baee2e564c2e013a32d69e/1656417839840/Guide+for+RDS-PS.pdf.

[11] "Industrial systems, installations and equipment and industrial products - Structuring principles and reference designation - Part 10: Power supply plants," IEC 81346-10, Standard, Aug. 2022.

[12] "Industrial systems, installations and equipment and industrial products - Structuring principles and reference designations - Part 2: Classification of objects and codes for classes," IEC 81346-2, Standard, Jun. 2019.

[13] E. Codd, "Further normalization of the data base relational model," *Courant Institute: Prentice-Hall.*, Oct. 1972.

[14] A. Beaulieu, *Learning SQL*. O'Reilly, 2009.

[15] PostgreSQL, "F.23. ltree — hierarchical tree-like data type," Dowloaded: 09.05.2024. [Online]. Available: https://www.postgresql.org/docs/current/ltree.html.

[16] M. Saikot, "What is the osi model – 7 layers of osi model explained," Nov. 2021, Accessed on May 1, 2024. [Online]. Available: https://bytexd.com/osi-model/.

[17] D. Crockford and C. Morningstar, "Standard ecma-404 the json data interchange syntax," Dec. 2017.

[18] W3Techs, ""usage statistics of javascript as client-side programming language on websites"," May 2024, Accessed on April 22, 2024.

[19] S. Kobourov, "Spring embedders and force directed graph drawing algorithms," Jan. 2012.

[20] D. Von Winterfeldt and W. Edwards, "Advances in decision analysis: From foundations to applications," in Cambridge University Press, 2007, pp. 81–103.

Appendix

The appendix is included as a .zip file attachment divided into three parts.

Appendix A

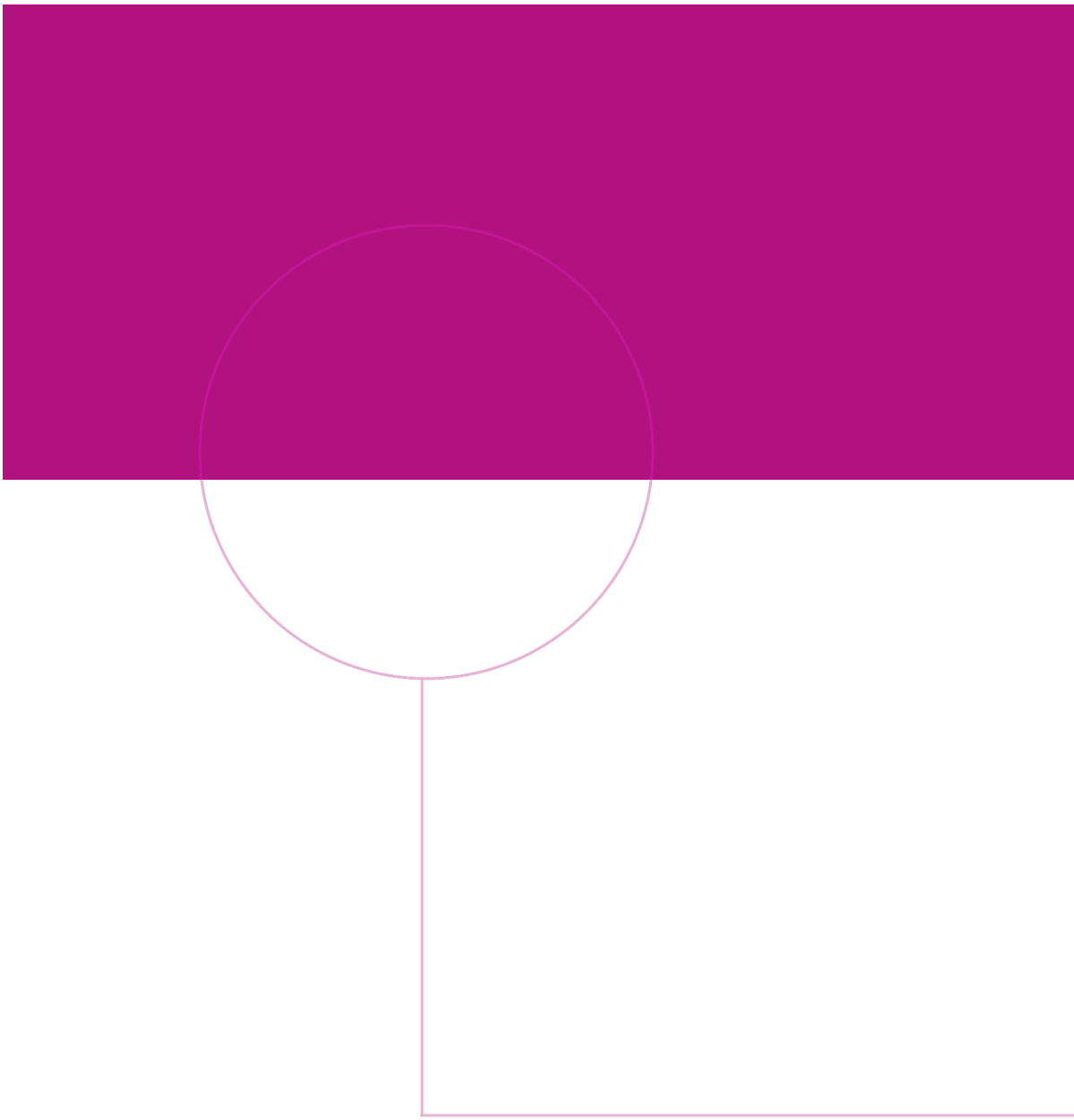
Appendix A includes all programming files, sectioned into front-end and the Go API.

Appendix B

Appendix B includes SQL database dump files that can be used to recreate the databases implemented in this project. A CSV file of the `object` table is also provided.

Appendix C

Appendix C contains the project poster.



NTNU

Norwegian University of
Science and Technology