Miika Karjalainen

Improving the Scalability and Performance of an IEC 61850-6 SCL Workflow

School of Technology and Innovations
Automation and Information Technology
Master of Science

Vaasa 2026

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Miika Karjalainen
Title of the thesis: Improving the Scalability and Performance of an IEC 61850-6 SCL Workflow
Degree: Master of Science (Technology)
Degree Programme: Automation and Information Technology
Supervisor: Tomi Pasanen
Year: 2026 **Pages:** 98

ABSTRACT:

The thesis addresses the modernization of a critical legacy tool used internally at ABB for generating IEC61850 Substation Configuration Language (SCL) configuration data for protection relays, along with matching C code. The tool is nearly 20 years old and serves a critical role in the build process and design of protection and control functions. It suffers from significant technical debt, which mostly relates to the usage of inefficient custom-implemented data structures and the monolithic architecture, which lacks the separation between user interface and business logic.

The custom-implemented data structures employ linked lists with deep nesting. The structure makes the search operations $O(n)$ making the build times using this tool take multiple hours. The long build times make the tool unfeasible for modern software development with fast iteration cycles.

During the build process of SCL data and the C code for a sample configuration, the tool produces around 7000 warnings and automatically fixes some errors in the source files. This creates uncertainty in the workflow as new warnings might not be seen, or they might not be seen as serious.

This study follows a constructive research approach with a performance evaluation and architectural mapping of the legacy system. The modernization process will implement a new cross-platform solution with an improved architectural approach, with separation between the user interface and business logic. The data handling is improved by introducing hash-based collections in the form of an N-ary tree with hash map children referred to during the thesis as a dictionary trie. The error-prone logic is replaced with XSD-generated C# classes, reducing manual maintenance effort while ensuring strict compliance with IEC 61850-6 SCL format.

With the implementation of a dictionary trie in the new implementation, the tool performs 369.1x better during the parsing of all sample configuration-related SCL files, as the tool's search times are no longer bound by the size of n. For the search operations the thesis manages a speedup of 14,165.6x at a matching worst-case level. As the amount of data is expected to rise in the future, the speedup factor is expected to increase.

KEYWORDS: substation configuration language, data structures, search algorithms, benchmarking

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen akateeminen yksikkö
Tekijä: Miika Karjalainen
Tutkielman nimi: IEC 61850-6 SCL -työnkulun skaalautuvuuden ja suorituskyvyn parantaminen
Tutkinto: Diplomi-insinööri
Koulutusohjelma: Automaatio ja tietotekniikka
Työn ohjaaja: Tomi Pasanen
Valmistumisvuosi: 2026 **Sivumäärä:** 98

TIIVISTELMÄ:

Tässä opinnäytetyössä käsitellään ABB:llä sisäisesti käytettävän kriittisen *legacy*-työkalun modernisointia. Työkalua hyödynnetään IEC61850 Substation Configuration Language (SCL) -konfiguraatiodatan sekä vastaavan C-koodin generointiin suojareleelle. Työkalu on lähes 20 vuotta vanha ja sillä on kriittinen rooli suojaus- ja ohjaustoimintojen suunnittelussa sekä koontiprosessissa. Työkaluun on kertynyt huomattavaa teknistä velkaa, joka liittyy pääasiassa tehottomien, räätälöityjen tietorakenteiden käytöstä sekä monoliittisesta arkkitehtuurista, jossa puuttuu käyttöliittymän ja liiketoimintalogiikan välinen erottelu.

Räätälöidyt tietorakenteet hyödyntävät syvästi sisäkkäisiä linkitettyjä listoja. Tämä rakenne tekee hakutoiminnoista $O(n)$ -aikaisia, minkä vuoksi työkalun avulla suoritetut koontiajat kestävät useita tunteja. Pitkät koontiajat tekevät työkalusta epäkäytännöllisen moderniin ohjelmistokehitykseen, jossa hyödynnetään nopeita iteraatiosyklejä.

Esimerkkikonfiguraation SCL-datan ja C-koodin koontiprosessin aikana työkalu tulostaa noin 7000 varoitusta ja korjaa automaattisesti joitakin virheitä lähdetiedostoista. Tämä luo epävarmuutta työnkulkuun, sillä uusia varoituksia ei välttämättä huomata tai niitä ei ehkä pidetä vakavina.

Opinnäytetyö käyttää konstruktiivista tutkimustapaa, johon sisältyvät suorituskyvyn arviointi ja vanhan arkkitehtuurin kartoitus. Modernisoinnissa toteutetaan uusi alustariippumaton ratkaisu ja parannettu arkkitehtuuri, jossa käyttöliittymä ja liiketoimintalogiikka on erotettu toisistaan. Tietojen käsittelyä parannetaan ottamalla käyttöön hajautukseen perustuvia kokoelmia N-asteisen puun muodossa, jonka solmut sisältävät hajautustauluja. Työssä tähän rakenteeseen viitataan termillä *dictionary trie*. Virhealtis logiikka korvataan XSD-skeemasta luoduilla C#-luokilla, mikä vähentää manuaalista ylläpitotyötä ja varmistaa samalla tiukan yhteensopivuuden IEC61850-6 SCL-formaatin kanssa.

Uuteen toteutuksen *dictionary trie*-rakenteen ansiosta työkalu suoriutuu 369,1 kertaa paremmin kaikkien esimerkkikonfiguraatioon liittyvien SCL-tiedostojen jäsentämisessä, sillä työkalun hakuajat eivät ole enää riippuvaisia joukon n koosta. Hakutoimintojen osalta tässä työssä saavutetaan 14165,6-kertainen nopeutus vastaavalla pahimman tapauksen tasolla. Koska datamäärän odotetaan kasvavan tulevaisuudessa, nopeuskertoimen oletetaan myös jatkavan kasvua.

AVAINSANAT: substation configuration language, tietorakenteet, hakualgoritmit, suorituskykyvertailu

Contents

Figures

Tables

Code

Abbreviations

IED – Intelligent Electronic Device

MV – Medium Voltage

SCL – Substation Configuration Language

XML – Extensible Markup Language

FD – Function Design

LD – Logical Device

LN – Logical Node

DO – Data Object

DA – Data Attribute

DUT – Device Under Test

TDD – Test-Driven Development

PDD – Performance-Driven Development

DOM – Document Object Model

XMI – XML Metadata Interchange

MMS – Manufacturing Message Specification

GOOSE – Generic Object-Oriented Substation Event

MU – Merging Unit

SMV – Sampled Value

LC – Logical Connection

XSD – XML Schema Definition

SA – Substation Automation

DOI – Instantiated Data Object

SDI – Instantiated Sub data

DAI – Instantiated Data Attribute

BDA – Basic Data Attribute

ICD – IED Capability Description

IID – Instantiated IED Description

SED – System Exchange Description

SCD – System Configuration Description

CID – Configured IED Description

UI – User Interface

SRP – Single Responsibility Principle

ACT – Application Configuration Table

PGO – Profile-Guided Optimization

JIT – Just-in-Time

AOT – Ahead-of-Time

1 Introduction

ABB develops intelligent electronic devices (IEDs) that are used in various environments to protect and control a full range of medium-voltage (MV) power distribution applications (ABB, 2025a). Typical sectors where these devices are deployed include the utility sector, which covers electric power companies, industrial plants handling oil and gas, manufacturing, and infrastructure, such as data centers and marine installations. The devices are used to control and protect sectors, detect faults, and automatically disconnect to prevent damage, such as fire.

These IEDs are built in compliance with International Electrotechnical Commission (IEC) 61850 Edition 2.1 and Edition 1, which enables modern substation automation. IEC 61850 series offers a way to build multi-manufacturer interoperable power utility automation (IEC, n.d.-a). IEC 61850 utilizes a data model approach based on the Substation Configuration Language (SCL), which is an Extensible Markup Language (XML) -based format defined in IEC 61850-6. The format describes IED configurations, parameters, communication system configurations, switch yard structures, and the relations between them (IEC, 2024).

To construct the SCL-based model and the corresponding C code for an IED, an internal tool is used within ABB. The tool can be used for function designs (FDs), edit existing ones, and convert them into SCL data and C code. Currently, the tool is used to construct over 170 functions with multiples of some, resulting in a total of 769 functions. As the utility grids evolve, the need for more functionality rises, and new functionality is added constantly in the form of new function designs or duplicate instances of already existing function designs.

Functions are the fundamental building blocks of an IED, which have a specific functionality or an algorithm. These functions follow the IEC 61850 model of logical nodes (LN). The functions have some specific usage, such as PHLPTOC, which is a three-

phase non-directional overcurrent protection low stage. Multiple instances of each function may exist in the IED, such as for PHLPTOC in REX615, where there exist 3 instances: PHLPTOC1, PHLPTOC2, and PHLPTOC3 (ABB, 2024).

Furthermore, ABB adds some private element extensions within the SCL data. The extensions contain information that is used by the Protection and Control IED Manager (PCM600) configuration tool. It shows data related to the application configuration tool, which is used to select what functionality the relay should measure and what it should act on.

The tool is used for each device within the ABB Relion product offering and is responsible for the productization phase within the build process. As the tool generates the C code and the SCL data model, the productization phase determines what functions to include, their quantity, and their appropriate default values if needed.

1.1 Problem Statement

The tool was developed approximately 20 years ago, when ABB did not have such large amounts of functionality, and the scalability of the tool was not considered, as the data model consisted of just a few logical nodes. Today, the current set of functionality of a device under test (DUT) 1 produces over 500,000 rows of SCL data, whereas DUT 2 produces over 1,000,000 rows of SCL data in addition to the generated C code for all the function designs.

The legacy solution that handles the IEC 61850 data model does not scale with the customer demands for more functionality, making the development cycle slow and resulting in significant waste of developer time. As ABB moves to a faster release cycle, the internal tool can’t meet the needs of the faster iteration cycles. The IED software must be built on a nightly basis in addition to feature builds that are triggered by the developers independently. The slowness affects the other parts, creating a bottleneck in the nightly pipeline, as waiting delays the test pipelines, which wait for the finished build.

The tool introduces uncertainty in the workflow by automatically attempting to fix certain errors in the input files instead of failing out early and prompting the developer for a fix to the input files. As the tool finished generating the final SCL model and C code, it resulted in over 7000 warnings for the DUT 2 product configuration. Blinded by the 7000 warnings, the developer may not notice any new warnings or errors that might occur when implementing new functions or improving old functions.

As the internal tool is used to construct over 170 functions into C code and SCL data, along with the previously mentioned extensions, it takes a local development machine around 2h 40min to build. The slow build time is mainly due to the inefficient data structures within the tool for the SCL data, as well as the private extensions and the algorithms that are used to search and mutate the data within them.

The tool relies largely on a custom implementation of linked lists, which nest around 4 layers deep for the SCL data, and the data is even more complex for private extensions, which nest to 5 levels. The data structures force the system to perform searches through the tree-like data structure for each retrieval operation. To search for a specific node, the tool starts from the topmost node, performing string comparisons until it finds the node. In the form of computational complexity, these structures result in the average case $O(n)$ search time.

Finally, the codebase suffers from architectural anti-patterns such as heavily tying Windows Forms (WinForms) user interface directly into the data model logic. Because of the tight coupling, a refactor would be a highly complex process. As the UI components are tied into the data classes themselves, it makes it harder to implement tests for the data model without touching the UI components. The current system also has no automated unit tests to catch any possible programming errors, so each iteration must be manually tested by comparing the tool outputs from a prior version and the current version to identify any possible regressions.

1.2 Objectives

The thesis's primary objective is to replace the legacy tool with a modern, high-performance, and modular implementation that can meet the demands of faster release cycles at ABB. Modularity ensures that each software component within it can be reused in future projects. Furthermore, the solution should be easily unit testable by developers to ensure confidence needed to make changes and improvements in the code. To achieve testability, each component must not be directly tied to other components, therefore eliminating the legacy system's tight coupling of WinForms into the business logic.

The first objective is to address the performance bottleneck of the tool, which is caused by the heavy use of legacy custom linked-list data structures and the algorithms around the data structures. The new implementation targets to reduce the search operation time complexity $O(n)$ average case by replacing the linked lists with hash-based data structures. With the upgraded data structure, the aim is to reduce the current 2h 40-minute build time to at least under 5 minutes.

The modernized tool should be able to handle data models that are significantly larger than the current DUT without significant degradation in performance, which depends on the data input size n. The only exceptions are the final code generation and writing of the full combined product XML file, which remains $O(n)$ operation as these operations must iterate over the whole data model to write them.

The implementation does not attempt to refactor the legacy codebase and instead attempts to recreate the software from the ground up. Refactoring the codebase would prove to be much more challenging, as retrofitting unit tests to the existing codebase would require a near-complete rewrite itself. The objective with the complete rewrite is to enforce a strict architectural separation between responsibilities to ensure long-term maintainability.

1.3 Methodology

This thesis follows a constructive research approach, following a seven-step process defined by Lukka (2003). According to Lukka, constructive research is a methodology that is applied to real-world problems where the goal is to gain a deep understanding of an existing problem and produce a practical artifact, such as a model, method, or implementation, which is applied to a real problem. The usage of constructive research is deemed a good fit in software engineering and information systems research, where theoretical contributions and practical solutions are often developed together and validated against real constrained problems.

As outlined by Lukka (2003), the process starts by identifying a relevant problem that has the ability to produce a theoretical contribution. Following the identification, the second phase is to examine the potential for long-term research with the target organization. For this thesis, ABB has shown commitment to resolving the friction within the workflow that is caused by the legacy tool.

After the problem foundation is set, the Lukka defines the third step to obtain a deep understanding of the problem, practically and theoretically. To get a complete understanding of the software, it is reverse engineered to understand its inputs and outputs, and what transformations the software does to the SCL data, as well as the generated code based on it. At ABB, a set of coworkers has been selected to help in understanding the software's nuances as well as validating decisions throughout the project.

Lukka (2003) defines the fourth phase as focusing on the innovation and how the solution solves the problem, while the fifth handles the implementation of the solution and the testing of the solution using a practical test. The result is tested by a subset of its users as a practical test. In addition to the practical tests, the solution is tested using benchmarks that measure the performance of the implementation under different

conditions. As real innovations are done on the data model and parsing approaches, the computational benchmarks highlight the abilities and performance of the data model implementation. The benchmarks use real SCL data to measure the performance of the implementation, along with synthetic data that is generated to stress test the implementation to prove its scalability in the long term.

Finally, Lukka (2003) defines the sixth step to determine the scope of applicability, which is to analyze the results of the process and determine if the construction might be transferable to other organizations if it produces the anticipated results. Finally, to conclude the study, the theoretical contribution is analyzed.

2 Literature Review

The public literature on SCL-specific tooling is quite sparse, likely because most tools are proprietary. However, some relevant works directly mentioning SCL data do exist, such as “Research on IED Configurator Based on IEC 61850” (Pan, 2011). IEC 61850 SCL Validation Using UML Model in Modern Digital Substation (Jang et al., 2018). Despite these studies, there remains a gap in the literature regarding the optimal data structures and architectural solutions for scalable and high-performing SCL workflows, particularly with handling large-scale hierarchical data.

The challenge is mostly focused on processing massive hierarchical XML files with keyed data. Additionally, it involves implementing a data structure to handle the SCL data. Therefore, pivoting to generic data structures and XML parsing approaches, the amount of literature broadens.

2.1 Parsing approaches for XML Data

The underlying XML data can be handled with multiple different approaches, such as using a document object model (DOM), which is commonly implemented in the .NET ecosystem with XmlDocument and XDocument classes, which construct a full in-memory representation of a given XML data. Streaming-based approaches, such as XmlReader in .NET, allow for forward-only parsing with low memory overhead. Finally, object mapping, which is also referred to as serialization, enables transformation between XML and object-oriented data.

The parsing approaches are directly referenced in Busatto et al. (2005) which highlights that the DOM representation is 4-5 times larger in the physical memory than the original XML file. XML parser performance is also directly measured by Nicola and John (2003) which measures the performance of different XML parsing methods, as well as DTD and XML schema validation, to measure the overhead.

2.2 Hash-Based and Hierarchical Data Structures

The nature of SCL data is its keyed hierarchical namespace, where elements can be identified uniquely with a structured path. Multiple solutions exist for handling structured data, such as using linked lists, where each node maintains a collection of child elements, which is exactly the approach used in the legacy tool. Tree structures are the natural way to represent XML documents due to their hierarchical nature, although typical tree implementations typically store children in sequential collections, which is not optimized for key-based lookup that is used in the legacy tool workflow. The SCL data model within the tool must be mutable so that a data node can be inserted at any position within the model, provided prior nodes exist. The data model must maintain the hierarchy of the data, as enumeration of the data model must be possible, and nodes must be searchable within the data model.

Two implementations of data structures are relevant to this thesis, with the implementation of the dictionary trie data structure. The dictionary trie takes advantage of the keyed nature of SCL data by utilizing hash tables while preserving the hierarchy by using a trie structure.

For keyed data, hash tables are a useful approach that work by hashing the key to a slot index using a hash function and storing the data that is accessible by the key. The benefit of hash tables is that with a key, the location can be computed without scanning the collection. Cormen et al. (2022) directly mention that “hashing is an extremely effective and practical technique” with the benefit of $O(1)$ access time on average. Cormen et al. (2022) also give a comprehensive analysis of the performance as well as collision handling strategies like chaining and open addressing.

Hierarchical data, like SCL data, the trie, also known as the prefix tree data structure, presented by Fredkin (1960), where each cell holds the address to the next register of a fixed-size array of cells, where one cell per member is reserved for each alphabet, with a termination marker. The trie decomposes a key into a sequence of symbols and uses

symbols to select a child node, which results in a data structure with a search time of $O(L)$, where L is the length of the key.

2.3 Existing approaches to SCL data handling

Pan (2011) has done research on IEC 61850-6 SCL language and also discusses the large amounts of data, and also describes it as “too big to be processed by common means” and declares the standard complicated, which makes the configuration difficult. For parsing XML-based SCL data Pan (2011) also opts to use C# and implements an improved XML parser with C#, and defines SAX and DOM as unsuitable to handle SCL files due to their large size, and using DOM parsing would result in inefficient memory usage. As conclusions, they report a 22MB SCL file taking the new improved parser 20 seconds to load and occupying less than 75MB.

Jang et al. (2018) are more focused on the validation of the IEC 61850-6 SCL files using a UML model, however, they do define a relational memory database, which is of interest for this thesis in view of the data model. A compilation of a visual UML model is done and exported into XML metadata interchange (XMI) format, which is then categorized and inserted into the relational database.

Jang et al. (2018) also described the usage of the relational database model as flexible and capable of removing redundancies and ambiguities from the data model. Furthermore, claiming that the use of relational databases has a strong drawback of its inability to support schema evolution and a weakness to mismatches between object-oriented and relational databases. Use of a non-relational database is described as a potential improved model.

Using a relational database provides an improved average search time complexity of $O(\log N)$ compared to the $O(N)$ of nested linked lists assuming the database is an implementation of B-tree like structure. The database has the worst-case scenario of $O(n)$. The average search time complexity would be an improvement over the legacy

approach used in the tool. However, given the tool constraint of writing the structure back to the file, the usage of a relational database has poor hierarchical preservation, which requires expensive JOIN operations on the database to reconstruct the original hierarchy of the data, which would complicate the process of writing all the data back. Using a relational database and having to reconstruct the original hierarchy makes it unviable for the use case of producing a complete XML file, as well as code generation, as these are highly dependent on the hierarchy of the data.

While not directly literature, an open-source solution worth mentioning exists for handling IEC 61850-6 SCL data, such as libiec61850 which supports parsing of SCL files into objects and saving them back (MZ Automation GmbH, 2017/2026). The library uses an implementation where lists are nested in a hierarchical way.

While research exists on general XML data structure performance, to the best of the author's knowledge, no prior literature can be found discussing the optimal data structures directly used for IEC 61850-6 SCL data, and this is identified as a research gap. Most previous work seems to focus on using relational databases, lists, and linked lists, and on applying a structural mismatch to the SCL. These solutions overlook the keyed nature of IEC 61850-6 SCL data and utilizing dictionaries in a hierarchical way. This thesis aims to solve the gap with the use of hash-based data structures and a parser that can fill the SCL in-memory data model, while valuing fast search times and hierarchical preservation with a write-back capability.

3 Technical Background

This chapter presents the technical background that is necessary to understand the problem domain and its constraints. The first part covers a subset of IEC 61850 standard, such as general principles from the first part, basic communication structure, and information from parts 7-1, 7-2, 7-3, and 7-4, and the configuration description language for communication in power utility automation systems related to IEDs from part 6, which explains the data format that the thesis works with.

The second section focuses on algorithm analysis and data structure theory, providing essential information about time complexity analysis as well as space complexity analysis. A strong focus is put on hash tables due to their importance in the new data structure implementation that holds the SCL data efficiently, time complexity-wise.

3.1 IEC 61850 Standard

IEC 61850 is a large standard that consists of 10 parts (Figure 1). The standard is very comprehensive as it defines communication protocols, the IEC 61850 data model, the SCL language, and security features which are based on the IEC 62351 series (IEC, n.d.-b).

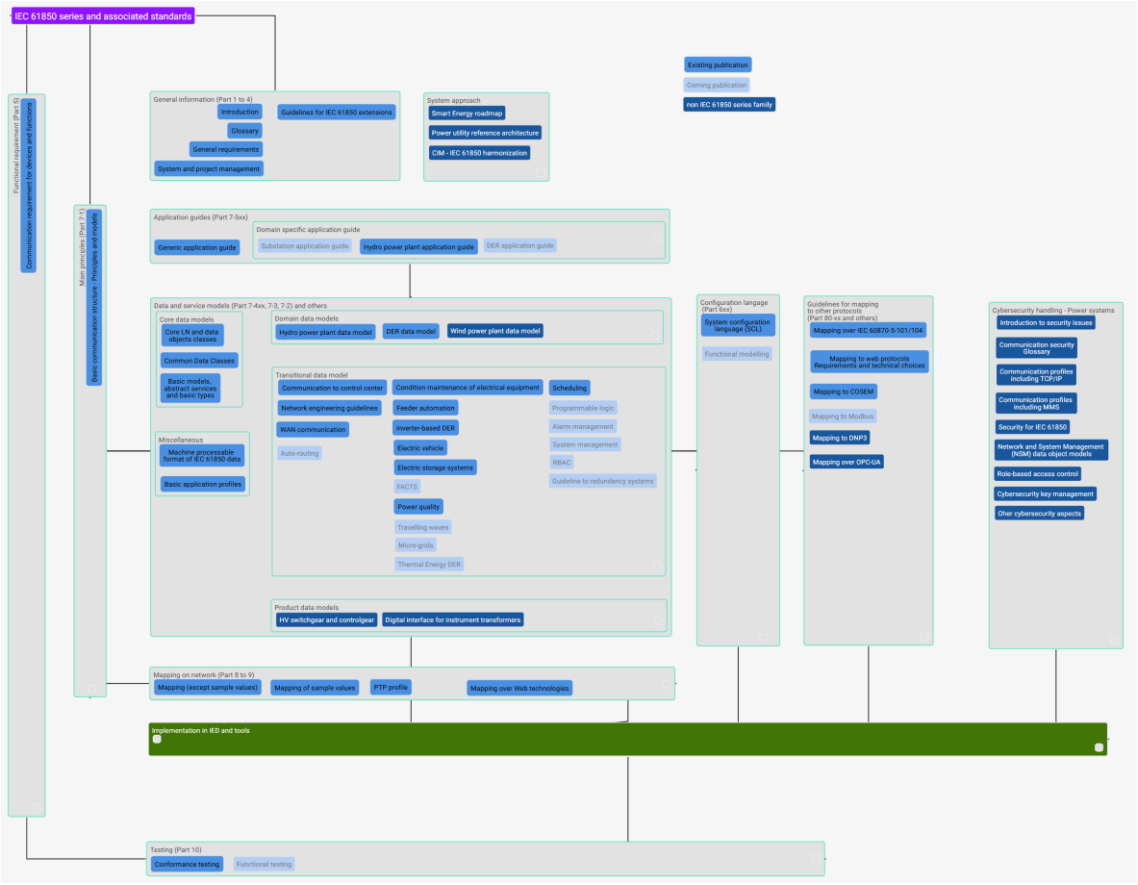

Figure 1. IEC 61850 Standard Map (IEC, n.d.-c).

The standard has been created by the IEC technical committee 57: Power systems management and associated information exchange. The first edition was published from 2002 to 2005, and the second edition, which supersedes the first edition, was published from 2009 onwards. (IEC, 2013, p. 13)

According to IEC (2013), before the adoption of IEC 61850, the industry relied on manufacturer-specific proprietary communication protocols instead of a standardized protocol. The lack of standardized protocols resulted in the need for costly protocol converters when using IEDs from various manufacturers. Examples of such legacy protocols include the proprietary ABB SPA-bus, Modbus, and DNP3. According to Newton-Evans Research (2019) Modbus and DNP3 are still widely used the most in North American markets within substation automation, 60% are using DNP3, and 19% are using Modbus.

IEC 61850 allows the devices to communicate with ethernet communication. To communicate between devices, the standard defines Manufacturing Message Specification (MMS), Generic Object Oriented Substation Event (GOOSE), and Sampled Values (SV) (IEC, 2020b, 2020c). Primary benefits of using IEC 61850 communication are its interoperability between different manufacturers, allowing engineers to integrate IEDs from several vendors, and preventing platform lock-in (IEC, 2013, p. 12).

3.2 IEC 61850 Data Model

IEC 61850 is split into multiple data models, namely core data models, domain data models, transitional data models, and product data models (IEC, n.d.-c). The core data model is defined within the standards IEC 61850-7-2, IEC 61850-7-3, and IEC 61850-7-4. IEC 61850-7-2 contains the basic model, abstract communication service interface (ASCI), and basic types, IEC 61850-7-3 expands upon this by adding common data classes (CDC), and finally, IEC 61850-7-4 adds compatible logical node and data classes (IEC, 2013).

As defined by IEC (2020a, p. 21), the conceptual model of IEC 61850, the basic model, abstract services, and basic types in IEC 61850-7-2 can be split into two parts: a meta-meta model and a meta model. The meta-meta model contains the base types, generic data, attributes, nesting, and composition, which is used by the meta model that defines the logical nodes and common data classes (CDC) and attributes.

The meta model has a few information modeling classes, IEC (2020a, p. 21) defines them as follows: “server represents the external visible behavior; logical device (LD) represents the information produced and consumed by a group of domain-specific application functions; logical node (LN) contains the information produced and consumed by a single domain-specific application function.”

Furthermore, the IEC (2020a) defines that parts 7-3 and 7-4 contain the domain type model. The domain type model contains defined types of logical node classes defined in

part 7-4 and defined types of common data classes in part 7-3. The data instance model then describes instances of the classes with the substation configuration language (SCL) as defined in IEC 61850-6.

The standard defines that IEC 61850 applies type definitions to all the nodes, which are categorized into basic types and domain types. According to IEC (2020a, p. 33) basic types contain all the non-structured and non-domain types, such as signed and unsigned integers, floats, and booleans. The domain types, on the other hand, contain abstract types, non-structured domain types, enumeration types, coded enum types, structured types, packed list types, and arrays.

The enumerated types represent an ordered set. An enumeration type allows for custom values within the element, but limits its usage to 127 characters, and each enumeration must follow a naming scheme of the title and an appended “Kind” (IEC, 2020a, p. 33). These types are marked into the bType element attributes when referenced in IEC 61850-6 SCL data.

To filter and find specific types of functionalities, a functional constraint is used within data attributes. The standard includes the following functional constraints: status information (ST), measurands (analogue values) (MX), setting (SP), substitution (SV), configuration (CF), description (DC), setting group (SG), setting group editable (SE), service response (SR), operate received (OR), blocking (BL) and extended definition (EX) (IEC, 2020a).

The data model uses a hierarchical data model approach. The root container is the IED, which represents a physical device, such as hardware and an operating system. Within a single IED, multiple logical devices can exist. The logical devices are defined as an information modeling class, which IEC (2020a, pp. 18, 21) furthermore defines as an “entity that represents a set of typical automation, protection or other functions” and as “represents the information produced and consumed by a group of domain-specific

application functions." Logical devices can be named as desired if they follow the naming conventions defined in IEC 61850-7-2 although within a logical device, a mandatory logical-node-zero (LLN0) must be defined (IEC, 2020a).

As an example of logical devices, DUT 1 has four. The DUT 1 contains LD0, which contains the protection-related logical nodes such as protection algorithms, measurements, and self-supervision signals, a CTRL instance that defines control functions for manual and automated switching operations, a DR instance containing disturbance recorder-related logical nodes, and a MU01 instance, which provides merging unit (MU) functionality for sampled values (SMV).

In the standard, IEC (2020a, pp. 18, 22) each logical device is further split into logical nodes, which represent individual typical automation, protection, or other functions. Inside the logical nodes, data objects (DOs) store information such as quality, timestamps, and position. Logical nodes can then be connected by using logical connections (LC) to exchange data between the nodes, as each node may require data from another (Figure 2).

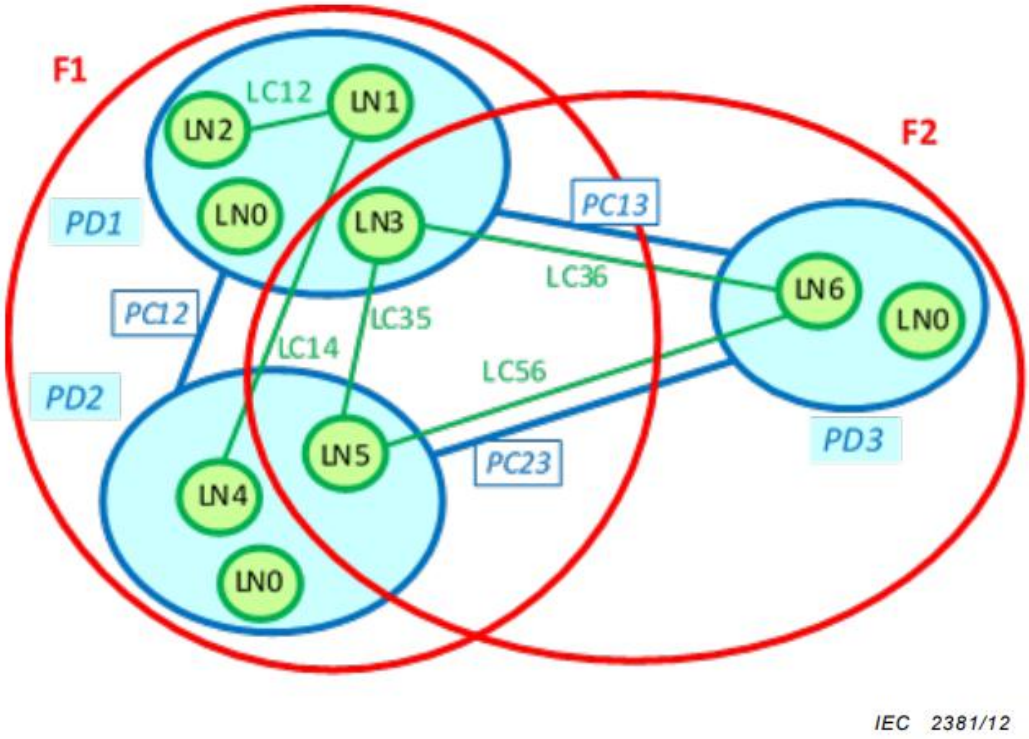

Figure 2. Logical nodes and logical connections (IEC, 2022, p. 34).

Finally, data objects contain data attributes (DAs), which can be nested within sub-data objects (SDOs) or be positioned directly under the data object. Data attributes are the final members in the hierarchy that contain the actual value.

3.3 Substation Configuration Language

According to the IEC (2013, pp. 13–14), the substation configuration language (SCL) is a file format used for the IEC 61850 standard. The file format defines the format for defining IED configurations, parameters, communication systems, switch yard structures, and relationships between them. The SCL format is based on the W3C Extensible Markup Language (XML) version 1.0. IEC ships an XML schema (XSD) with the IEC 61850-6 standard, which has the SCL namespace. The namespace is identified with the following string: "`http://www.iec.ch/61850/2003/SCL`" (pp. 13–14).

The IEC (2024, p. 18) defines the scope of SCL as being focused on providing a data format for substation automation system (SAS) functional specification, IED capability description, and substation automation (SA) system description with support for exchanging system information between two projects or two systems, and the ability to read data of IED modifications on an IED instance back to the chosen configuration tool.

Furthermore, IEC (2024, pp. 26, 29–32) defines that the SCL object model consists of four parts: process, product, communication, and a data type template section. The primary process can be a substation, a line, or a process. Its responsibility is to describe primary process-related functions and devices. The product is for all substation automation-related objects, including IEDs. The communication part is for all communication-related objects, which can contain subnetworks, access points, and connections between IEDs. Finally, a data type template section is included, which describes reusable types for used data objects, data attributes, and logical nodes.

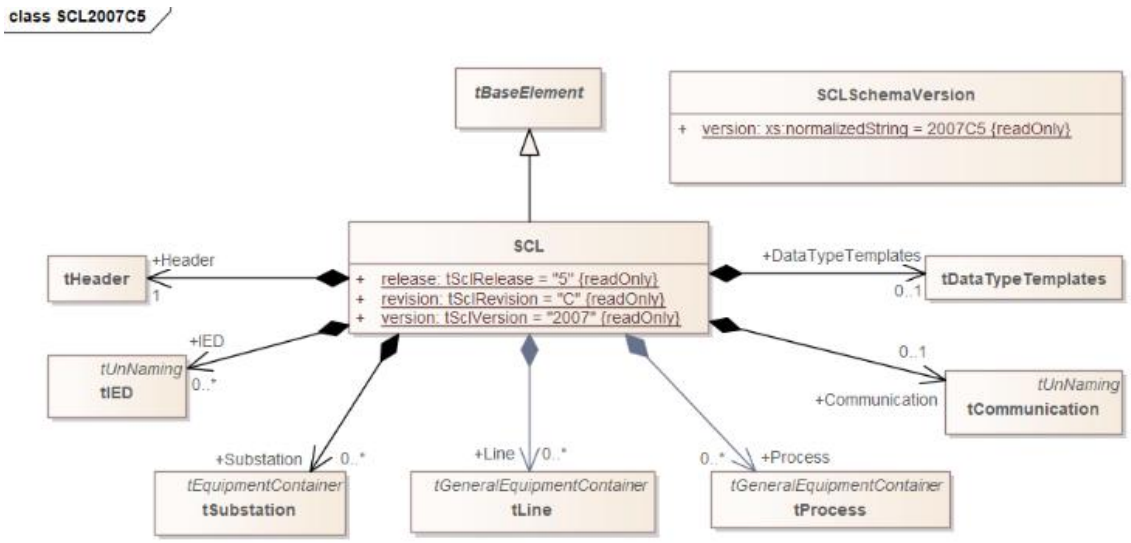

Figure 3. UML diagram overview of SCL schema (IEC, 2024, p. 37).

Looking at Figure 3, the SCL format defines the process and line elements, which are the responsibility of the PCM600 tool and the device connectivity package. The process and line elements are part of the process model (IEC, 2024, pp. 29–30).

3.3.1 IED Description

As detailed by IEC (2024), the product model covers the IED elements (Figure 4) that construct the substation system. Within an IED element, a server section exists that describes the communication system containing the logical devices, authentication, and association elements. Logical nodes (Figure 5) are placed within the logical devices. Each logical node then contains its respective data objects and data attributes. As seen in the UML descriptions in figures 4 to 6, the IED section is quite deep and highly hierarchical. For a single IED, this section takes a significant chunk of the data within the SCL file.

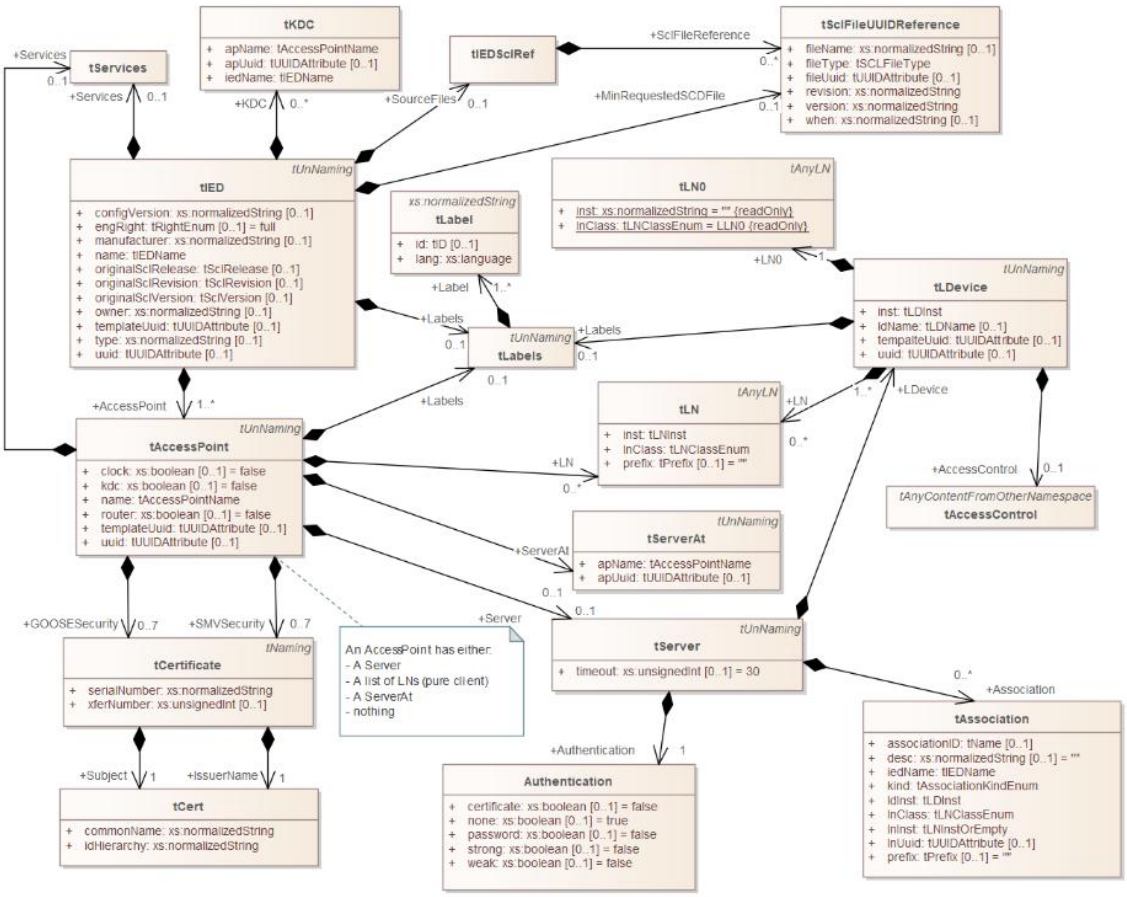

Figure 4. UML description of IED-related schema part (IEC, 2024, p. 88).

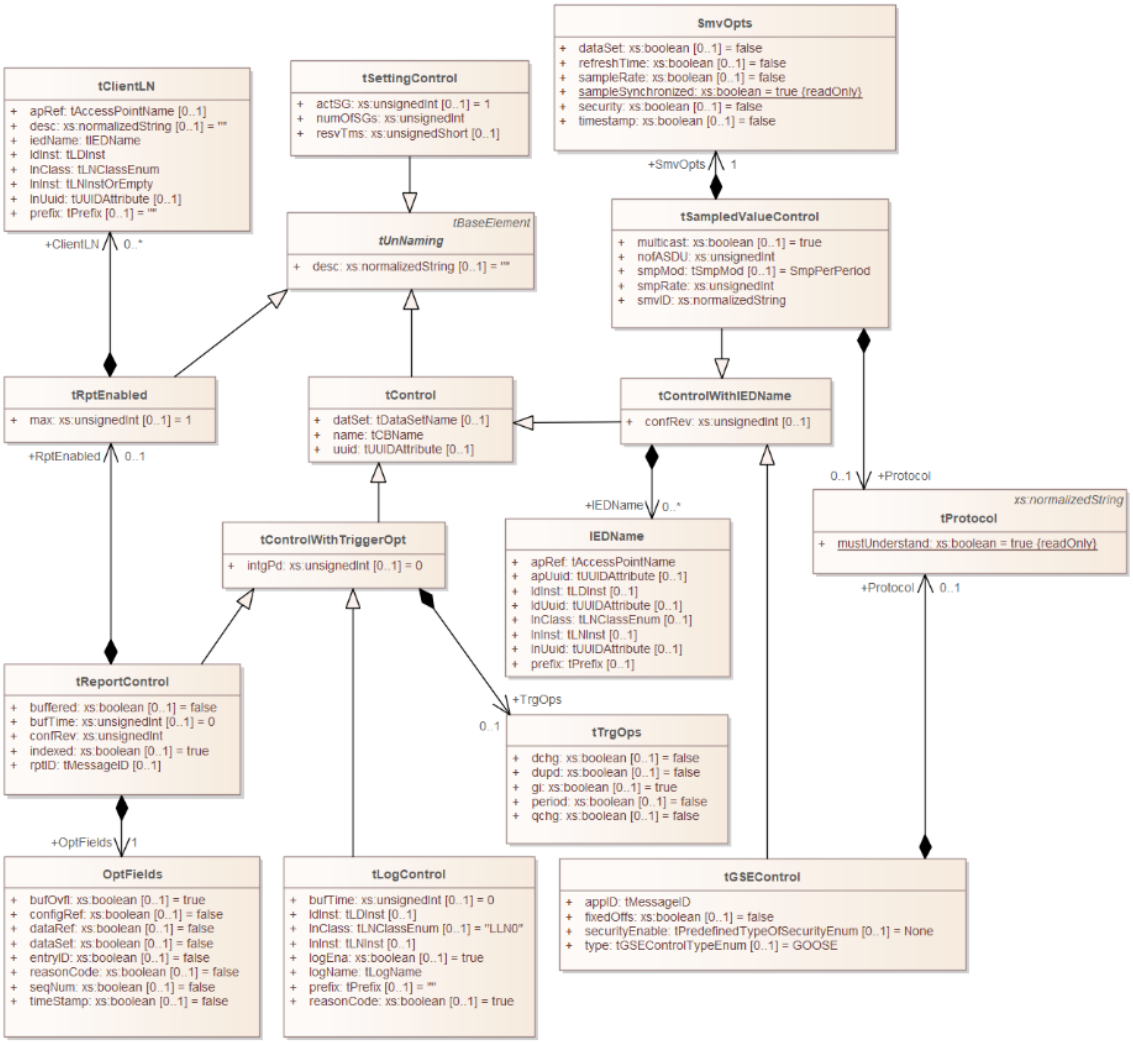

Figure 5. UML description of IED-related schema part for control blocks (IEC, 2024, p. 89).

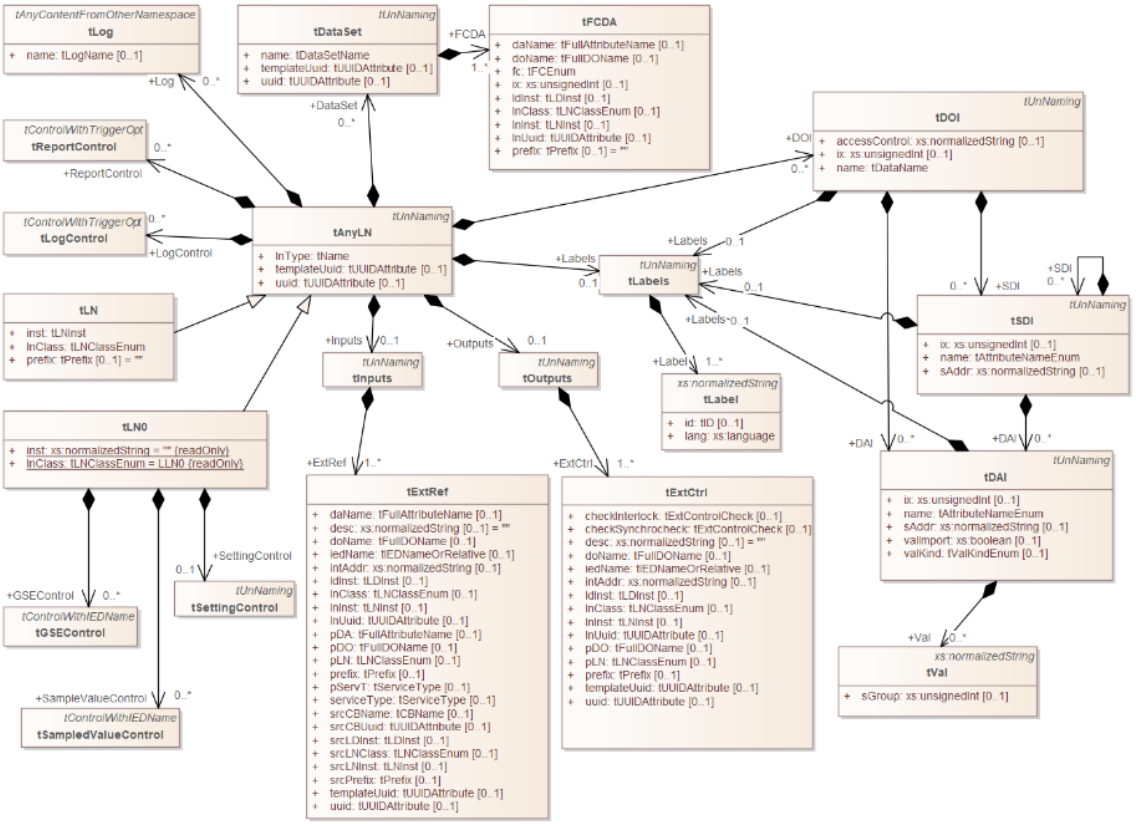

Figure 6. UML description of IED-related schema part (LN definition) (IEC, 2024, p. 90).

The standard defines that an IED object may have a Router function, which is used to connect different subnetworks using its access points. An IED router can be defined without any logical devices or logical nodes, meaning that it's managed from a separate network (IEC, 2024, p. 86).

For each IED, only one IED section must represent the IED (IEC, 2024, p. 86). Moving to the definition of the IED, the standard has some restrictions, such that the IED must be unique within the SCL and empty values are not permitted. Additionally, the maximum allowed length for the name can be a maximum of 64 characters. In case the IED is a template, the name of the IED must be "TEMPLATE" (p. 94).

As defined within the IED, logical devices exist that are marked with the LDevice element. To identify the logical devices, the inst attribute is used and must be unique within the IED. The standard defines that LDevice also contains a description attribute which can be

used for further information, an ldName attribute defining the name of the logical device within communication, an optional unique identifier (UUID), and an optional template UUID (IEC, 2024, pp. 109–110).

IEC defines that (2024, pp. 110–113) LDevice must contain the LN0, which is a special logical node that does not need an inst attribute, as the inst attribute for LN0 is always LLN0. Other logical nodes within the logical device must define an inst, lnClass, and prefix attributes, and optionally UUID fields for the logical node and a possible template UUID. Logical nodes also define an lnType, which is a reference to a LNodeType definition.

According to IEC (2024, pp. 114–116), data objects are placed within each logical node and identified as DOI elements. Within each DOI element, either a DAI element or a substructure name part SDI element exists. The SDI element is used as a structure element that can contain more SDI elements or a DAI element, which ends the branch. DOI has a description attribute “desc” which can be used to define further information, and a name attribute that follows the IEC 61850-7-4. The name field is used for resolving the type defined in the LNodeType definition. In the case of an array type, an ix field defines the index of a data element. Finally, an “accessControl” attribute, which defines the access control for the data.

Furthermore, according to IEC (2024, p. 115) the branch ending DAI also defines a description field for further information, a name which is used for type resolving purposes in the same way as for DOI, a short address field “sAddr”, valKind which defines the meaning of the value from engineering, ix which defines the index of the array if DAI is of array type, valImport which declare that the configurator supports import of values that have been modified by other tools.

Together, as defined in IEC (2024, pp. 47–48) these blocks form a complete path to a signal, which is referred to as a signal identification. The signal identification is built from 4 parts defined by some of the previously mentioned fields. The first part is the name of

the logical device, the second part is the logical node prefix, and the third part is the combination of the logical node class defined in IEC 61850-7-4 and the logical node number. The last part is the combination of the data name, as defined in IEC 61850-7-4, and the data attribute name, as defined in IEC 61850-7-3.

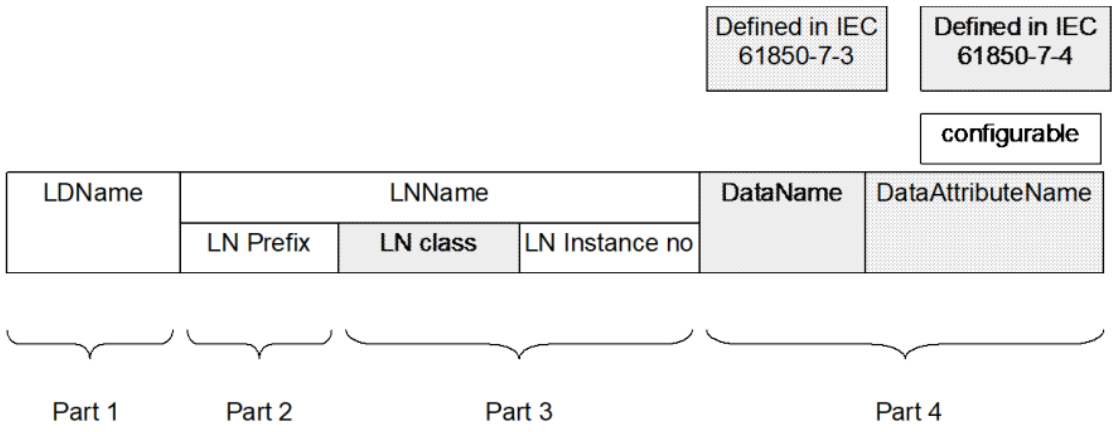

Figure 7. Elements of the signal identification (IEC, 2024, p. 48).

3.3.2 Communication Description

IEC (2024, p. 31) defines the communication model (Figure 8) which is a non-hierarchical model used to define possible connections between IEDs within subnetworks and across them. They are defined in the model as access points. The communication model also allows us to define “Router” types. Clock types are also allowed for defining a master clock for a certain subnetwork, which is then used to synchronize all clocks of all other IEDs connected to the same subnetwork.

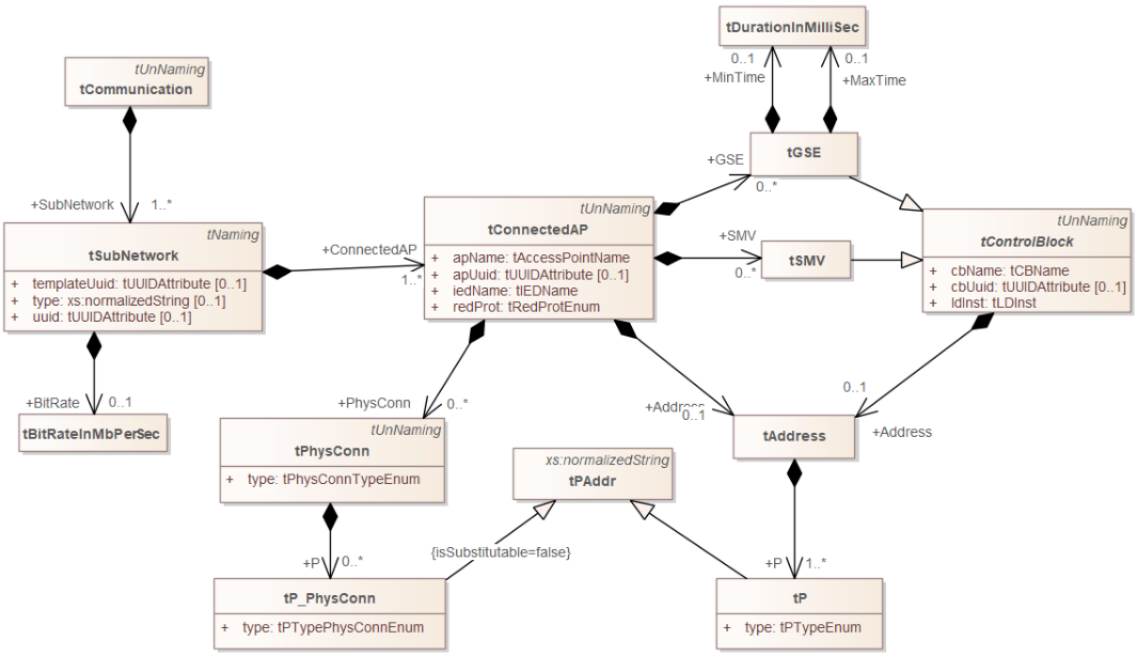

Figure 8. UML diagram overview of the communication section (IEC, 2024, p. 143).

The subnetwork defined by IEC (2024, p. 32) within the communication model is described for connecting nodes at the link layer (Layer 2) between access points. Devices in a subnetwork can talk to each other directly. An access point then connects the logical devices to a subnetwork. A single IED can have multiple access points that are connected to different subnetworks.

IEC (2024, p. 32) also defines a router bridges two subnetworks together. Where normally a device on a subnetwork 1 can only see servers on the subnetwork 1 router, functionality can extend across the boundary. A router only forwards services that use a networking layer and restricts the forwarding of layer 2 services.

The clock is the master clock for a subnetwork. The clock keeps the devices on the subnetwork in sync via a protocol like precision time protocol (PTP/IEEE 1588) or simple network time protocol (SNTP).

3.3.3 Data Type Template Description

IEC (2024, p. 152) defines a data type templates section (Figure 9) which purpose is to define the types for each logical node that can be instantiated. These are defined as LNodeType elements, and within each LNodeType element contains data objects (DO) elements. The data object elements within the logical node type elements describe the contents of a logical node.

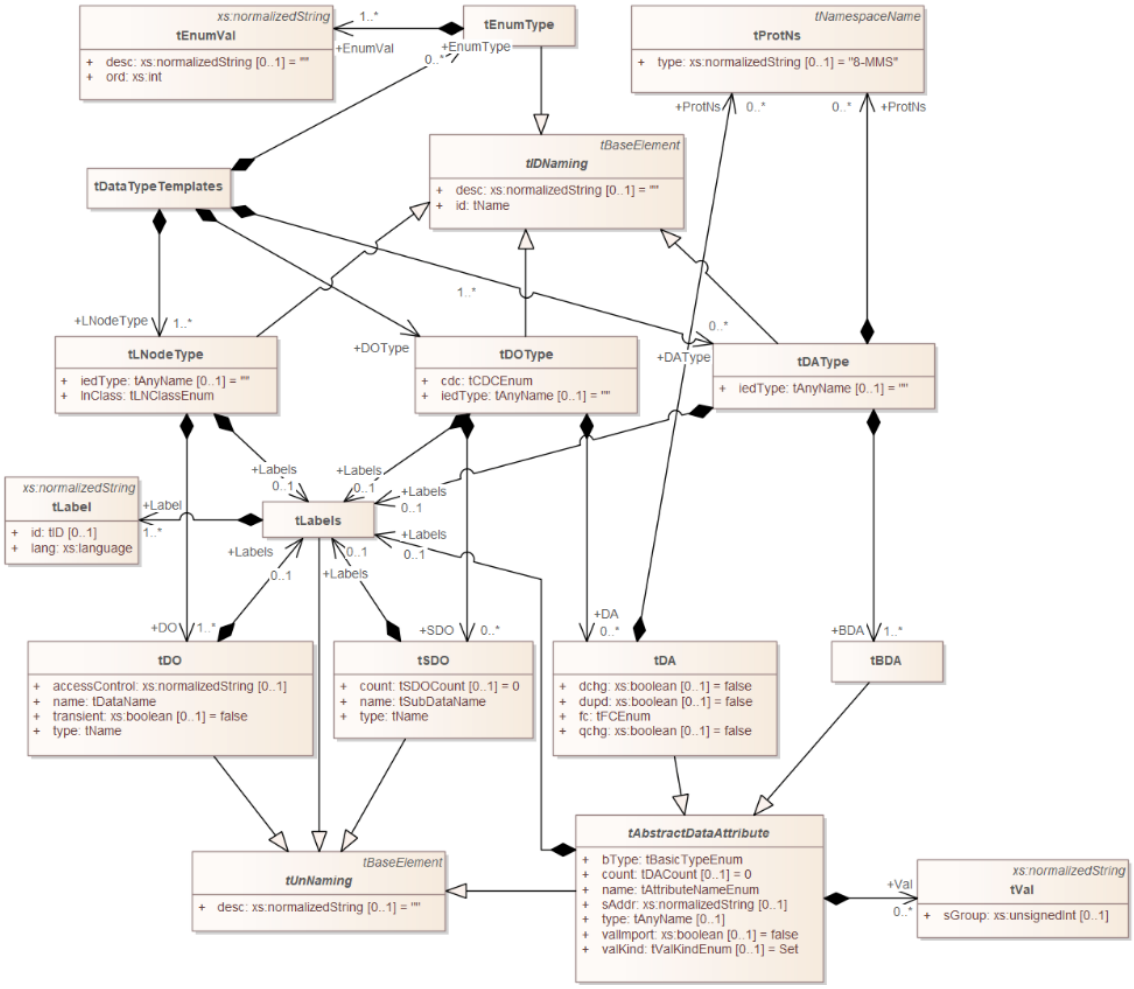

Figure 9. UML overview of the DataTypeTemplate section (IEC, 2024, p. 153).

Continuing the same standard, IEC (2024, p. 157) defines a LNodeType which has four attributes: id, which is used to identify the LN type within the SCL; desc, which can be used to provide additional information about the LN type; deprecated iedType field,

which defines to which IED the LN type belongs; lnClass specifies the base class of the type. In Code 1. LNodeType from a REX615 is shown for a PHPTOV1 instance. Within the LNodeType, several data objects can be identified with differing unique names, such as “Mod” and “Beh,” etc.

```
<LNodeType id="PHPTOV1_ED2_G_4" lnClass="PTOV">
   <DO name="Mod" type="ABBIED600_Rev3_ENC_Mod_OffOn_FD"/>
   <DO name="Beh" type="ABBIED600_Rev10_ENS_beh"/>
   <DO name="Blk" type="ABBIED600_Rev1_SPS_simple"/>
   <DO name="Str" type="ABBIED600_Rev2_ACD_threephase"/>
   <DO      name="Op"      type="ABBIED600_Rev2_ACT_threephase"
transient="true"/>
   <DO name="TmVCrv" type="ABBIED600_Rev6_CURVE_SG_Sub6"/>
   <DO name="StrVal" type="ABBIED600_Rev8_ASG_SG_i"/>
   <DO name="TmMult" type="ABBIED600_Rev8_ASG_SG_i"/>
   <DO name="MinOpTmms" type="ABBIED600_Rev2_ING_SP_1"/>
   <DO name="OpDlTmms" type="ABBIED600_Rev2_ING_SG"/>
   <DO name="RsDlTmms" type="ABBIED600_Rev2_ING_SP_1"/>
   <DO                                              name="TypRsCrv"
type="ABBIED600_Rev3_ENG_SG_TypRsCrv_Sub1_e"/>
   <DO                                                 name="NumPh"
type="ABBIED600_Rev4_ENG_SP_StrPhSel_Sub1_e"/>
   <DO name="StrDur" type="ABBIED600_Rev8_MV_2_e"/>
  <DO name="TestPro" type="ABBIED600_Rev8_ENC_TestPro_Sub2_e"/>
   <DO name="CrvSatRl" type="ABBIED600_Rev4_ASG_SP_f_e"/>
   <DO name="VSel" type="ABBIED600_Rev4_ENG_SP_VSel_Sub1_e"/>
   <DO name="HysRl" type="ABBIED600_Rev8_ASG_SP_i_e"/>
   <DO name="TypTmRs" type="ABBIED600_Rev2_ENG_SG_TypTmRs_e"/>
</LNodeType>
```

Code 1. Example LNodeType from REX615 SCL (ABB, 2025b).

IEC (2024, p. 152) specifies that each data object element then has a reference to a data object type (DOType) element. The DOType’s then contain data attributes (DA) or DOTypes (SDO) which have already been defined. A data attribute can either be a basic type, an enumeration, or a structure of a DAType. In Code 2, a DOType “ABBIED600_Rev3_ENC_Mod_OffOn_FD” is defined, which is used in Code 1, within the “Mod” data object. The DOType contains all the needed data attributes for the said type. Some DA elements specify a custom type, such as for data attribute Oper, the specified type is “ABBIED600_Rev1_tcOper_Mod_OnOff”.

```
<DOType id="ABBIED600_Rev3_ENC_Mod_OffOn_FD" cdc="ENC">
   <DA        name="Oper"        fc="CO"        bType="Struct"
type="ABBIED600_Rev1_tcOper_Mod_OnOff"/>
   <DA         name="stVal"         fc="ST"         bType="Enum"
type="ABBIED600_Rev1_BehaviourModeKind_OnOff" dchg="true">
         <Val>on</Val>
   </DA>
   <DA name="q" fc="ST" bType="Quality" qchg="true"/>
   <DA name="t" fc="ST" bType="Timestamp"/>
   <DA        name="ctlModel"        fc="CF"        bType="Enum"
type="ABBIED600_Rev1_CtlModelKind_StatusDirect"    dchg="true"
valKind="RO" valImport="false">
         <Val>direct-with-normal-security</Val>
   </DA>
</DOType>
```

Code 2. Example DOType from REX615 SCL (ABB, 2025b).

DATypes are then built from basic data attributes (BDA) elements (IEC, 2024, pp. 152–153). In Code 3, the type “ABBIED600_Rev1_tcOper_Mod_OnOff” is shown, which contains multiple basic data attributes. The BDAs can also specify types such as structures or enumerations.

```
<DAType id="ABBIED600_Rev1_tcOper_Mod_OnOff">
   <BDA                name="ctlVal"                bType="Enum"
type="ABBIED600_Rev1_BehaviourModeKind_OnOff"/>
   <BDA              name="origin"              bType="Struct"
type="ABBIED600_Rev1_originator"/>
   <BDA name="ctlNum" bType="INT8U"/>
   <BDA name="T" bType="Timestamp"/>
   <BDA name="Test" bType="BOOLEAN"/>
   <BDA name="Check" bType="Check"/>
   <ProtNs type="8-MMS">IEC 61850-8-1:2003</ProtNs>
</DAType>
```

Code 3. Example DAType from REX615 SCL (ABB, 2025b).

LNodeType, DAType, DOType, and EnumType are in the DataTypeTemplate sections, where each element has an id field, a string used to identify each type. The id field must be unique within all IEDs (IEC, 2024, pp. 152–153).

3.3.4 Private SCL Elements

The IEC (2024, pp. 42–43) allows the use of manufacturer-specific extensions within the data by using a “Private” element. Data marked by private tags is preserved during data exchange between tools. The preservation must happen in a way that its location in the file matches the original location of the private data.

According to the IEC (2024, p. 43) these private data elements must be contained within their own explicit namespace. Each private data is stored within a private definition and must use a manufacturer-specific string part. The private element has a type attribute for identification purposes (IEC, 2024, pp. 43–44).

These extensions can be used to create data that’s used by certain manufacturers like ABB. ABB uses private extensions to generate configuration tool-specific data for application sheet data, which is a function drag and drop system within the ABB configuration tool PCM600.

Code 4 contains an example of ABB common substation automation data, which is used to define functions and function groups. Code 5. contains an example of the drag-and-drop block that is used by the PCM600. The tACT is a private XML namespace that defines the inputs and outputs that are used by the configuration tool to know which inputs and outputs each function block has. These function blocks are representations of functions that the relay implements, such as a switch controller SCSWI.

The private extensions can become problematic with larger amounts of data, as the XSD schema does not enforce uniqueness within them. Some other design flaws within them are that the root nodes have no identification other than their type, meaning that, for identification purposes, the parser must proceed further to find their type. The data also duplicates some data that is already defined within the non-private SCL data in elements such as the logical nodes.

```
<commonSA:Object type="Function">
       <commonSA:Name>CTRL_CBCILO1_Control</commonSA:Name>
                      <commonSA:Node            name="CBXCBR1"
id="CBXCBR1" descId="Circuit breaker control" fbRevision="H"
guid="6e1cf178-6008-404c-9145-f0952278c0f6"/>
                      <commonSA:Parameters>
                           <commonSA:Parameter
name="CTRL.CBCSWI1.Pos.stVal"   guid="0375fc77-7c04-4951-a518-
d106af2dbb1c">

   <commonSA:Value>0</commonSA:Value>

   <commonSA:CaptionID>CSWI_Pos_stVal</commonSA:CaptionID>

   <commonSA:DescID>CSWI_Pos_stVal_DESC</commonSA:DescID>
                                  <commonSA:Values
dataCategory="IOData" type="singleChoice">
                                  <commonSA:Enum
textID="STD_Dbpos_0" value="0"/>
                                  <commonSA:Enum
textID="STD_Dbpos_1" value="1"/>
                                  <commonSA:Enum
textID="STD_Dbpos_2" value="2"/>
                                  <commonSA:Enum
textID="STD_Dbpos_3" value="3"/>
                                  </commonSA:Values>
                                  <commonSA:Access
category="basic">Read</commonSA:Access>
                           </commonSA:Parameter>
                      </commonSA:Parameters>
                      <commonSA:Menu              id="CBXCBR1"
name="CBXCBR1">
                           <commonSA:Menu  id="CSWI_Pos_stVal"
name="POSITION" link="CTRL.CBCSWI1.Pos.stVal"/>
                      </commonSA:Menu>
</commonSA:Object>
```

Code 4. Example of Common Substation automation object data (ABB, 2025b).

```
<tACT:FB     Name="MINMAXAVE12R"     Max="10"     cat="Logic"
fbRevision="A"     EGD="*"     guid="6618e168-319f-4da6-9947-
c41d02112cbf">
                  <tACT:IN       Name="REAL_IN1"       Pin="1"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="af037e73-
dde5-46ef-8dd6-892aff7c686e"/>
                  <tACT:IN       Name="REAL_IN2"       Pin="2"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="980c4b0f-
a54c-4d08-8389-46b769797dac"/>
                  <tACT:IN       Name="REAL_IN3"       Pin="3"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="d6d7f986-
3640-453f-baa7-44d4a9eb2d0c"/>
```

```
                <tACT:IN       Name="REAL_IN4"       Pin="4"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="dc25b491-
d42d-4315-adbc-9c9104bad7c8"/>
                <tACT:IN       Name="REAL_IN5"       Pin="5"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="8023258e-
af06-44a7-a04b-1dc3dd21e14e"/>
                <tACT:IN       Name="REAL_IN6"       Pin="6"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="e23d0ff0-
3f2b-48e6-8c45-f628311da614"/>
                <tACT:IN       Name="REAL_IN7"       Pin="7"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="33a34dac-
5ca0-4912-a3c0-a8c710461c77"/>
                <tACT:IN       Name="REAL_IN8"       Pin="8"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="1808ca42-
2f82-42ef-8811-535a9d60f536"/>
                <tACT:IN       Name="REAL_IN9"       Pin="9"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="161de195-
6e82-48e6-9a63-61dc0cb33134"/>
                <tACT:IN      Name="REAL_IN10"      Pin="10"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="60a1432c-
be14-4e4a-8a77-b209dab50139"/>
                <tACT:IN      Name="REAL_IN11"      Pin="11"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="09c14e41-
baca-4303-b6fb-957d8f4dbe11"/>
                <tACT:IN      Name="REAL_IN12"      Pin="12"
Type="FLOAT32" view="ACT" Default="-2147483648" guid="b88c77d4-
fc74-4591-88ec-06e721ce35ae"/>
                <tACT:OUT  Name="MIN"  Pin="1"  Type="FLOAT32"
view="ACT"      Default="0"      guid="30fa6b3e-9594-45cf-97b8-
af4abb142ed4"/>
                <tACT:OUT  Name="MAX"  Pin="2"  Type="FLOAT32"
view="ACT"      Default="0"      guid="8a6c7550-0c2b-4925-b84c-
296a9e164cfa"/>
                <tACT:OUT  Name="AVE"  Pin="3"  Type="FLOAT32"
view="ACT"      Default="0"      guid="9485d1fc-deda-4888-9788-
04be39211033"/>
                <tACT:OUT Name="VALID" Pin="4" Type="BOOLEAN"
view="ACT"      Default="0"      guid="761b9804-87a8-4eb4-a3f0-
50bff630546b"/>
</tACT:FB>
```

Code 5. Example of ACT function block data (ABB, 2025b).

3.3.5 Engineering Workflow

As the SCL data is used in the engineering process IEC (2024, pp. 19, 171–172) dictates the usage of two engineering tools: a system configurator and an IED configurator. These tools can also be combined into a single tool. The IED configurator's purpose is to modify the data model, parameters, and configurations for a single IED instance. The system

configurator then handles the creation of multiple IEDs and manages how the devices connect and communicate within the project.

To handle SCL data IEC (2024, p. 21) has defined multiple file types. These are defined for each part of the engineering process for identification purposes. To configure an IED, an IED capability (ICD) file containing the IED capabilities is imported from the IED database to the system configurator. The IED can also be instantiated using an instantiated IED description (IID) file. The ICD files can be generated in advance, and an actual connection to the IED may not be needed, allowing the engineer to configure the IED beforehand.

The system configurator then generates an SCD file, which is imported by the IED configurator for IED instance configuration. Values added by the IED configurator can then be exported using an IED description (IID) file back to the system configurator. (IEC, 2024)

3.4 Algorithm and Data Structure Analysis

According to Cormen et al. (2022, pp. 86–87) to analyze algorithms and data structures, asymptotic efficiency is used to calculate the running time of an algorithm, which are shown using an asymptotic notation. For data structures, algorithm analysis applies to the time and space complexity of insertion, deletion, and search operations within the data structure.

The authors explain that different notation types are used depending on the bounds of the asymptotic efficiency. O-notation shows the upper bound of the asymptotic efficiency of an algorithm, also known as an asymptotic upper bound, meaning that the algorithm, “grows no faster than a certain rate, based on the highest-order term” (Cormen et al., 2022, pp. 87, 91–92). Ω-notation is used to show a lower bound of the asymptotic efficiency of an algorithm, meaning that an algorithm, according to Cormen et al. (2022, p. 87) “grows at least as fast as a certain rate as in O-notation on the highest-

order term.” Θ-notation shows the tight bound of the asymptotic efficiency of an algorithm. This means that the algorithm “grows precisely at a certain rate, based once again on the highest-order term” (Cormen et al., 2022, p. 87).

Time complexity analysis is done to determine the running time of an algorithm, depending on the input size of n. Time complexity is shown using the asymptotic notations above. For data structures, the time complexity is calculated for the most relevant operations, such as search, insertion, and deletion operations, as these are the most used operations for data-intensive applications. This thesis focuses in particular on search operations and their improved implementations, which are discussed in detail in later sections.

Space complexity analysis, on the other hand, describes the memory usage of an algorithm or data structure as a function of its input size n. Like time complexity, space complexity is also expressed in asymptotic notation.

3.4.1 Hash tables

According to Cormen et al. (2022) hash tables are a form of data structure that allows the average search time of $O(1)$ with the worst case of $\Theta(n)$ time. Hash tables use a hash function to compute the slot number in the table from the key. In practice, this works by defining an element with a key k. The key is then hashed with the hash function h to slot by $h(k)$.

Cormen et al. (2022) further note that using a hash function, two or multiple different keys might hash to the same slot, which is called a collision. When a collision happens, there are a few techniques to resolve it.

Furthermore, according to Cormen et al. (2022) the collision depends on the hash functions’ performance in creating unique hashes for the keys. To represent collision visually in Figure 10, a universe of keys U is defined where K contains the actual keys.

The keys map into the table T where $h(k)$ is the hash function. Both keys k_2 and k_5 map to the same slot in the table.

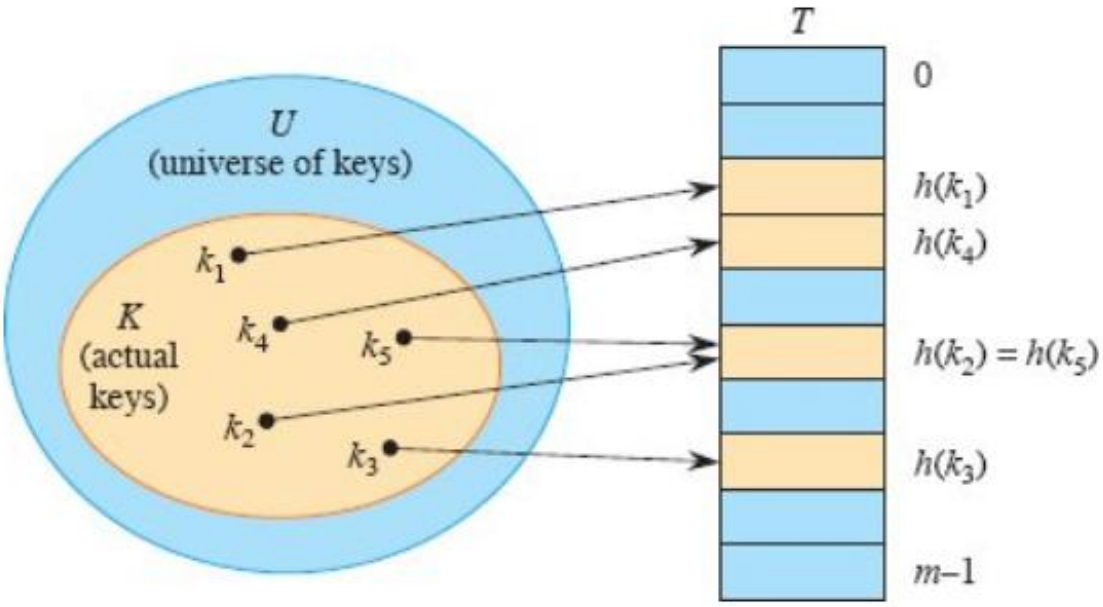

Figure 10. Collision with keys $\boldsymbol{k_2}$ and $\boldsymbol{k_5}$ (Cormen et al., 2022, p. 373).

To measure the likelihood of a collision, the Cormen et al. (2022, p. 204) show that birthday paradox can be applied to hash functions. A hash function with n possible output values and k items hashed the probability of a collision happening can be calculated approximately:

$$P(collision) \approx 1-e^{-k(k-1)/2n}$$

Using that formula with a given hashing function depends on the output space of the hash; for example, with an output space of 64 bits, we have the following possible hashes n:

$$2^{64} = 18446744073709551616$$

The 50% chance of a collision occurring, we would need the following number of different strings to reach:

$$5.05694 \times 10^9$$

Cormen et al. (2022) explain that collisions can be resolved by chaining. Each filled slot would point to a linked list, and in case of a collision, the value would go into the slot's linked list. Using chaining, the worst case for insertion is $O(1)$. For search operations, the worst-case is $\Theta(n)$ as you must go through the linked list. The average-case performance then depends on how well the hash function distributes the keys.

Furthermore, Cormen et al. (2022) state that with chaining, a load factor α is defined, which is calculated by dividing the number of elements n by the m number of slots for a given hash table. The performance in chaining is very dependent on the load factor, where the worst case $\Theta(n)$ given a large list of collisions, the performance is similar to a linked list.

$$\alpha = \frac{n}{m}$$

Another technique for resolving collisions is open addressing. According to Cormen et al. (2022, p. 394), instead of storing collisions outside, like in chaining, open addressing stores everything in the hash table. Open addressing works by inserting into the first location if it is not occupied, then trying to reach its second location. In open addressing, the load factor α must be less than or equal to 1 due to one element using each slot within the hash table. With a load factor over 1, the table would overflow when inserting new elements.

Inserting into a hash table that uses open addressing requires a probe. A common solution is to use linear probing. As described by Cormen et al. (2022, pp. 395–396), probing works by checking the table until an empty slot is found. Linear probing, for

example, works by, in the case of collision, checking the element right next to the occupied slot, and if it is free, the element is inserted there. Other probing solutions exist, such as quadratic probing and double hashing (Kuszmaul & Xi, 2024, p. 103:2).

In open addressing, deleting an element from the hash table must not simply set the element's position to null. Cormen et al. (2022, p. 396) explain that the reasoning behind this is that marking a slot as null when deleting breaks probing if any collisions have occurred. When deleting, the slot must be explicitly marked as deleted using a special value.

As a hash function is needed for a hash table, according to Cormen et al. (2022, pp. 380–381) it's important to choose a good one that approximately satisfies independent uniform hashing. There exist random hashing and static hashing functions. Static hashing functions use a single fixed hash function. Random hashing, on the other hand, uses a random hash function at random from a suitable family of algorithms.

According to Cormen et al. (2022, p. 381) random hashing is the preferred approach. Static hashing can be abused with attacks such as denial-of-service (DoS) known as hash flooding. If static hashing were used in applications processing unsafe user data, and the user knows the hashing algorithm, the hash table could be used to induce collisions. This takes advantage of the hash table's worst-case Θ(n) by making the size of n large through collisions.

Using open addressing instead of chaining also offers speed improvements through cache locality, as with open addressing, the blocks are stored in memory contiguously, unlike linked lists in chaining.

3.4.2 Multimaps

Baumstark and Pohl (2019, p. 210) describe the usage of multimaps which supports multiple elements having an equal key meaning that it supports a one-to-many

relationship. Regular hash tables use uniqueness among keys for each value in a one-to-one relationship, in some cases it is useful to map a single key to multiple values.

Multimap is often useful when there is no uniqueness between the keys, where multiple values map to the same key. Applying a regular hash table would result in discarding data or constructing keys to work around the constraint of non-uniqueness of the data.

A hash-based multimap has the same benefits as a hash table, such as the average-case time complexity for insertion $O(1)$. Search time complexity remains $O(1)$ for a whole list of returned nodes, but searching for a specific node inside the list is $O(1 + n)$ where n is the node's position within the list.

3.4.3 Trie (Prefix Tree)

Fredkin (1960) introduced a trie, also known as a prefix tree. Trie is a data structure that is used to store words of characters. It uses fixed-size arrays where each cell represents a character.

Fredkin (1960, p. 490) describes the model as each node in the trie is a register that holds a fixed-size array of cells, one cell per member of the alphabet. For alphabetic characters, including a space marker to indicate the end of a word, each register requires 27 cells in total. Each cell holds the address to the next register in the path.

The search time for a trie depends on the length of the key being searched. In asymptotic notation, it is represented as $O(L)$ where L is the length of the key. As an example, performing a search for a word like “APPLE” where L is 5 the data structure works by checking the root node “A”, then it moves to it and finds the next “P”, checks and moves to next “P”, checks and moves to “L”, and finally checks and moves to “E”.

The key property of the trie is its ability to store sequences of data. This can be applied to anything that shares a common prefix, which has a sequence already stored and does not require new storage space for the common data they share.

4 Analysis of The Current State

This chapter analyses the legacy SCL workflow that defines what the redesign must be able to achieve. The analysis begins with an architectural mapping of the tool to understand the structure and features of the tool and to show motivation for why the tool was completely rewritten instead of refactoring the code in the legacy solution.

The architectural mapping is followed by a review of previous attempts to increase the performance of the legacy implementation. An analysis is also provided, which identifies the performance bottlenecks that motivate the complete rewrite of the tool.

4.1 Architecture Mapping

The application features a broad set of features. It has a system internally called scripting, which is a command dispatcher system that allows the designation of workflows for specific products. The idea behind the scripting system is to automate the actions within the UI of the application without spawning the UI itself. The most important features of the tool are its code generation and the creation and modification of SCL data. Among other things, the tool aims to be an all-in-one solution for numerous other IED development-related functionalities in the development environment.

The command dispatcher system has a relatively simple syntax that allows setting variables, supports defining for loops, and simple conditionals. The dispatcher system pulls in a lot of data from different sources based on what is included in the script file. The system can traverse a graph of scripts and execute commands from other script files. Although the syntax used by the dispatcher is quite simple, multiple types of action calls exist. There are 225 actions within the system. These actions import data into the model, modify or delete data within the model by accessing certain fields across all nodes. Some are responsible for writing certain types of files, such as C code, complete XML, which combines all functions into one single file, outputting an HTML table of all data attributes.

The program can keep track of the current traversal level and regularly informs the user if an action is done on the incorrect scripting level. The scripting level system has been added to enforce separation between the script files, for example, warning the user if some data is imported at the 1st scripting level. However, this is largely ignored internally as it doesn't affect the generation in any way.

As data is imported, the software loads it into memory by using a custom implementation of linked lists that apply deep nesting following the tree-like structure of the data. Everything is added to the same model instance, where, at each addition to the tree, it looks up the correct position to add it.

To apply productization-specific values, a separate mapping file is used. The mapping files are defined using XML notation. At the top level, each mapping file contains a root-level mapping element and, within it, a list of nodes. These nodes in the mappings file can be addresses to a specific node by using their full name notation, or they can be invoked as a wildcard. Common uses of the wildcard are to set all instances that contain an attribute like stVal data attribute to an appropriate default value. Both operations start a tree traversal through the SCL data to find the correct node to modify.

To measure the technical debt of the tool, the assessment framework (Figure 11) by R. Marinescu (2012) is used to identify severe design flaws. The table features multiple design flaws such as god classes, code duplication, and intensive coupling.

Design flaw	Influence				Granularity	Severity
	Coupling	Cohesion	Complexity	Encapsulation		
God Class	Medium	Medium	Medium	High	Class	Number of data members used from other classes
Schizophrenic Class	Medium	High	Medium	Low	Class	Number of disjoint groups of clients using the interface of the class
Refused Parent Bequest	High	Low	Medium	Low	Class	Ratio of inheritance-specific members used from the superclass
Data Class	Low	Medium	Low	High	Class	Number of non-encapsulated data members and the number of other classes using the data.
Code Duplication	Low	Medium	High	Low	Method	Length of duplicated code and number of operations sharing that code
Brain Method	Medium	Medium	High	Medium	Method	Operation length and nesting level of statements
Data Clumps	Medium	Low	Medium	High	Method	Number of methods where the repeated parameters appear
Intensive Coupling	High	Medium	Low	Low	Method	Number of methods called from a single class

Figure 11. Impact of design flaws (Marinescu, 2012, p. 9:5).

Investigating the codebase in detail, the architecture reveals critical issues, such as god classes, which violate the single responsibility principle (SRP). Most application code is built from two instances of god classes with tight coupling, where the first one handles XML parsing, validation, code generation, file operations, gunzip operations, error logging, licensing, and the data model. The second object is a WinForms object that exposes a public field of the first object, creating the tight couple. Other classes also always reference the first god classes.

The application is full of intensive coupling everywhere. Most importantly, this makes unit tests extremely hard to implement, as mocking or stubbing dependencies is not possible. Other risks include that the tight coupling makes changes in one module risk breaking others. With the tight coupling, the flexibility is non-existent, meaning components can’t be changed to others easily without rippling everywhere.

As noted, with the current architecture, testing requires a lot of effort with manual testing of inputs and outputs, as unit testing is not implemented. The consequence is that the behavior cannot be verified during a refactor.

4.2 Previous Tool Improvements

The first implementations of the tool were using .NET Framework 4.8, which was later migrated to .NET 9. Migration to .NET 9 gave a significant performance boost with minimal code changes that only relate to removed WinForms components. The migration improved the execution time of the tool to 40 minutes. Migration to the newer version allows the use of modern C# language features, such as nullable and pattern matching.

The upgrade was done using a .NET Upgrade Assistant, which does automatic changes where it can, and the rest, where automatic changes can’t be applied, are given to the user to fix.

Visual Studio profiler was used to find hot spots in the program where the program spends most of its execution time. Several issues were discovered within the data structures and the algorithms that search them. Within the results listed in Table 1, string comparisons account for nearly half of the total CPU time, and the Object.FindFunction, Object.FindOutput and Object.FindInput is used to search the ABB private extension ACT data. These functions are called for each DAI within the data model to resolve if the type really needs to be in the final output, automatically correcting the user-made mistakes made in the source files.

Delving deeper into the usage of string comparisons within the software, a bunch of expensive search operations were found. The pseudocode for the found search functions is listed in Code 6 and visualized in Figure 12. The code searches through multiple layers of custom linked lists nested within each other resulting in $O(n)$ time complexity. Each time a new member is found, the string comparison is done to check if it matches the

one it is looking for. With this search function, if a node that doesn't exist in the data structure, it will go through the whole linked list's structure, visiting all nodes and comparing all string attributes.

With the way linked lists are implemented, the iteration to the next member will cause a pointer-follow, which jumps to a random memory address within the heap. This worsens the speed of the implementation as it affects the cache locality. Loading a single member fetches the cache lines, and the cache lines contain $X + 1, X + 2$ where X is the address. As the members are put into random addresses within the heap, they will not be included with the $X + 1$… cache lines, instead, it must access RAM to fetch the value.

The implementation does not implement any mitigations to address the cache locality, such as implementing a contiguous block so that all nodes are placed near each other in memory.

The code first iterates over the outer layer of logical devices, then logical nodes, then data objects, and finally data attributes. This results in the following asymptotic notation

$$O(n) = O(N_{LD} + N_{LN} + N_{DO} + N_{DA})$$

Where N_{LD} is the number of logical devices, N_{LN} the number of logical nodes, N_{DO} the number of data objects and N_{DA} the number data attributes.

All find functions searched through the nested linked lists, burning CPU time. To improve execution time, all calls to the add functions were identified to add an intermediary caching layer for faster lookups. The caching layer utilizes a hash-based dictionary type for $O(1)$ access time.

The use of an intermediary caching layer is justified by the program's complexity. The nested linked lists are quite hard to replace completely, as some methods manually search the model and do actions during iterations, whereas some other parts use the predefined available methods.

Table 1. Hottest functions after .NET 9 migration.

Function	**Total CPU [unit, %]**	**Self CPU [unit, %]**
System.String.op_Equality(string, string)	1292315 (46,01 %)	1232264 (43,87 %)
Object.FindFunction	1059157 (37,71 %)	547875 (19,50 %)
Object.FindOutput	729033 (25,95 %)	348254 (12,40 %)
Object.FindInput	610621 (21,74 %)	286805 (10,21 %)

```
foreach (var itemA in ListA)
{
    if(doesSubStringMatch(itemA.name))
    {
        foreach (var itemB in itemA.ListB)
        {
            if(doesSubStringMatch(itemB.name))
            {
                foreach (var itemC in itemB.ListC)
                {
                    if (itemC.Name == targetName) { /* Found
*/ }
                }
            }
        }
    }
}
```

Code 6. Nested search algorithm in legacy tool.

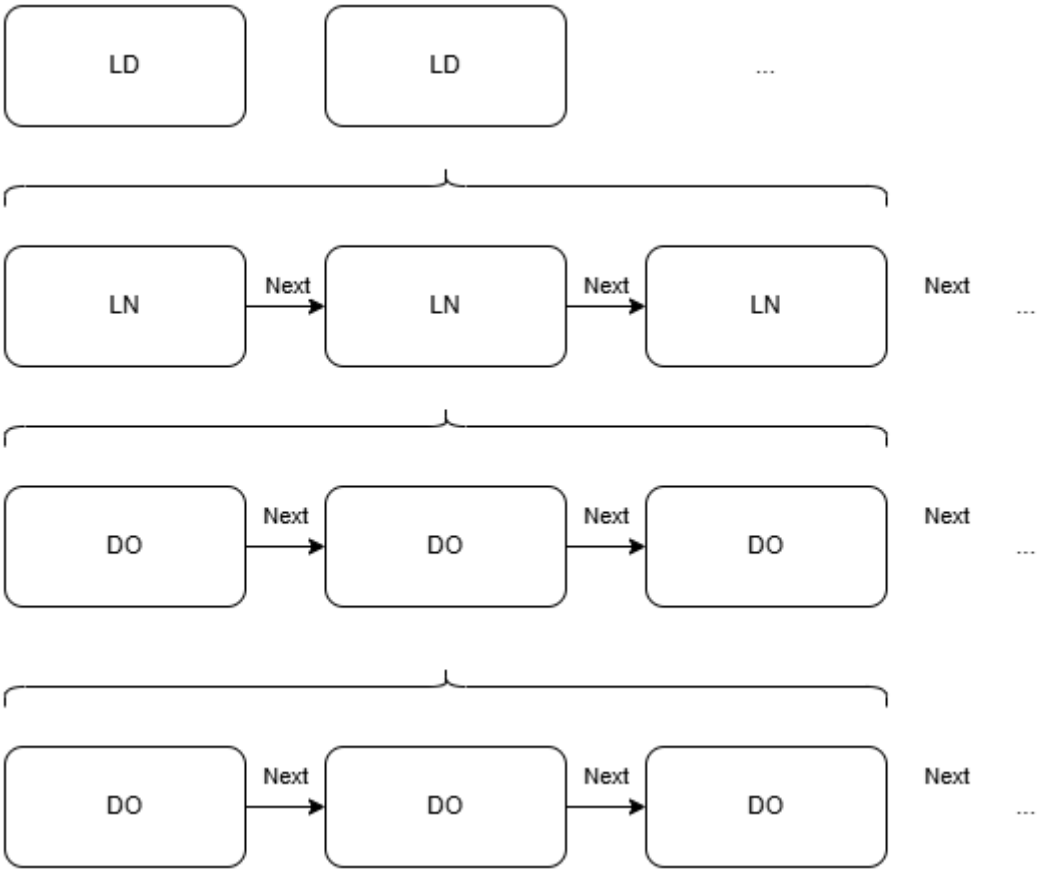

Figure 12. Legacy in-memory model using linked lists.

Separate instances of heavy functions were then invoked using BenchmarkDotNet in a separate project to obtain more specific information. BenchmarkDotNet allows the use of diagnosers, which can be attached to the benchmark to get additional information. To measure the impact of adding the caching layer, a benchmark is added that tests the performance of the dictionary layer.

Loading function instances currently in use within DUT 1 model, which contains 5 logical devices, 1141 logical nodes, 18811 data objects, and 212846 data attributes, resulting in a total of 232803 data nodes across 567 files. The original structure of four-dimensional linked lists results in the following time complexity with the DUT 1 model data.

$$O(n)$$

$$O(n_{LD} + n_{LN} + n_{DO} + n_{DA})$$

$$O(5 + 1141 + 18811 + 212846)$$
$$O(232803)$$

As seen in the above calculation, the worst-case scenario for searching the data model is 232803 operations. This case is even worse when loading all function instances in use within DUT 2, which offers the most functionality out of the product line. For DUT 2, it loads a total of 5 logical devices, 1,589 logical nodes, 28,102 data objects, and 320,951 data attributes across 769 files, resulting in the following worst-case access.

$$O(n)$$
$$O(n_{LD} + n_{LN} + n_{DO} + n_{DA})$$
$$O(5 + 1589 + 28102 + 320951)$$
$$O(350647)$$

So, with DUT 2 config, the worst-case scenario, where the node is the last, which we are looking for, results in a total of 350,647 operations. The disadvantage of using the four-dimensional linked lists is that they degrade linearly as more data is introduced. As seen, the size of n grows linearly as more data is introduced.

Usage of the intermediate dictionary improves the average-case search complexity to $O(1)$, the worst-case remains $O(n)$ where n is the number of nodes, so a total of 212,846 or 320,951. This holds only when we assume all instances collide, i.e., a full hash collision chain, which is unrealistic.

Using the birthday paradox defined by Cormen et al. (2022, pp. 202–204) to confirm the probability of a hash collision, for example, with a 32-bit output space, n is 4,294,967,296, and k is the count of hashed items, 320,951. We have the following approximate chance of encountering at least one collision:

$$P(collision) \approx 1 - e^{-\frac{k(k-1)}{2n}}$$
$$P(collision) \approx 1 - e^{-11.991851899074390530 59}$$

$$P(collision) \approx 99.99938055194673 2404\%$$

Further applying the balls-and-bins theory by Cormen et al. (2022, pp. 206–207) to see how many collisions the data will approximately have, we calculate the number of empty buckets $E[empty]$:

$$P(empty) \approx e^{-\frac{k}{n}}$$
$$P(empty) \approx 99.992527556409839573\%$$
$$E[empty] \approx n \cdot P(empty)$$
$$E[empty] \approx 4294646357$$

Using the number of empty buckets, we can calculate the number of occupied buckets, $E[occupied]$, and the number of collisions, $E[collisions]$:

$$E[occupied] = 4294967296 - 4294646357 = 320939$$
$$E[collisions] = k - E[occupied]$$
$$E[collisions] = 12$$

With the DUT 2 dataset, the number of collisions is just 12. This proves that even with this dataset, the time complexity remains $O(1)$ on average, and showing how unrealistic the full collision chain is with the data.

Part of the reason why the linked list tree performs so poorly is also due to the cache locality. As each operation is performed on the next and previous node in the linked lists, it forces the CPU to perform pointer chasing, leading to L1/L2 cache misses. When resolving pointers, the CPU must constantly pull data from RAM.

Dictionaries in C# are implemented in a way that utilizes buckets and slots that are placed in a contiguous array, which improves the cache locality. This improves the hit count as CPUs load parts of the nearby data when fetching from memory.

Table 2. Comparison of dictionary usage.

Method	**Mean (ns)**	**Error (ns)**	**StdDev (ns)**	**Ratio**	**Gen 0**	**Allocated (B)**
FindDaiRef_ByNameWithDict	100.5883	1.5407	1.4412	0.000	0.0274	344
FindDaiRef_ByNameNoDict	160,256.8394	1,932.8926	1,713.4589	0.032	-	96
FindDaiRef_ByNameWithDict_LastNode	113.3	0.81	0.75	-	0.0235	296
FindDaiRef_ByNameNoDict_LastNode	335,909.1	6,340.79	5,931.18	-	-	72

With benchmarking using the DUT 1 data model, data attributes are searched from the full model. In both benchmarking cases, the following node is called “LD0.LLN0.NamPlt.vendor”.

The method “FindDaiRef_ByNameWithDict” implements the new intermediary dictionary layer, which improves lookup times from 160,256.8394 ns to just 100.5883 ns. The queried element, however, exists very close to the head of the data model. When searching a known last node “LD0.WMMXU3.W.d”, the execution time is much higher at 335,909.1 ns without the intermediary dictionary, whereas with the dictionary it is 113.3 ns.

Each new insertion requires a linear scan to check whether the node already exists, unless the node has been added to the dictionary, in which case the linear scan is skipped. Usage of the dictionary forces the use of unique keys without any overhead, and insertion remains $O(1)$.

These statistics prove the usefulness of the intermediary dictionary layer within the application. With the intermediary layer in place, the lookups are greatly improved.
A downside of the intermediary dictionary layer is that it increases the memory footprint substantially, as object references are now stored in the tree as well as in the dictionary. This, however, is an acceptable trade-off for the speed benefits achieved.

5 Redesign and Implementation of the Tool

The new implementation also uses C# but a newer release of .NET version 10, allowing for high-performance implementation and cross-platform support. Development was done using Visual Studio 2026 with ReSharper and JetBrains Rider 2025.3. Several tools in Visual Studio and JetBrains Rider are used to analyze code and its performance. Within Rider, the embedded profiler dotTrace and the embedded decompiler are used.

Development was done using a comprehensive suite of tests, including benchmark and unit tests that verify the internal logic. The application is developed in a mixture of test-driven development (TDD) and Ashkin's (2019) model of performance-driven development. He defines performance-driven development as a 4-step process, starting with designing a task and its performance goals. A performance test is then written for the target method, and the code is created or modified as needed and checked in the new performance space. The usage of performance-driven development protects the developer from making any wrong optimizations.

The application is designed to be published with NativeAOT. NativeAOT is used to compile the C# code into native code instead of the C# intermediate language (IL). NativeAOT, however, comes with various limitations, most notably that the application can't use reflection, which adds some difficulties, and technologies to use must be selected carefully so that they are compatible with NativeAOT.

The architecture of the new application will position each component strictly into its own module. This improves the application's reusability, so that components implemented within this application can be used as modules for new tools and for improving old tools.

A completely new high-performance data model is created that takes ideas from the improvements made on the previous improvements, such as the usage of dictionary types.

The tool includes too many features that are not relevant to its main purpose, such as licensing and capability features. Such features are not implemented in the new tool at all and are implemented as separate scripts, which are outside the scope of this thesis.

Making the tool an all-in-one application makes refactoring code harder when it is performed. It is much easier to maintain a few scripts and a single tool separately than to try to implement all functionality in one, creating a complex, large codebase that requires much more help to onboard new employees.

5.1 Creating the Data Model

Creating a new data model to hold the SCL data must fill a few checks. As the data sizes vary per device from 500,000 lines to over 1,000,000 lines of SCL data, which consists of over 350,000 data attributes in the most demanding model. The quantity of the data needs to be considered when storing the data, to allow for fast lookups of a single node, and at the same time keeping the memory footprint sane.

The data model should also be able to handle ABB private extensions as well as the standard SCL data. However, the data model does not need to handle any unknown node types and is designed to reject any unknown private extensions that are not needed in the product. The data model should also be mutable, as default values for nodes might be changed per product.

Although Jang et al. (Jang et al., 2018) show the usefulness of a relational database for IEC 61850-6 SCL data usage. The idea of using an in-memory relational database like SQLite was rejected based on the highly structured nature of IEC 61850 data, which would require the expensive JOIN operations, as well as adding general overhead for SQL parsing and transaction management.

As used in the earlier improvements with the caching layer implemented into the legacy

application, which built a flat hash map of every node that was queried, this idea is also rejected here. The flat hash map is rejected since flattening the nodes would make the data lose its hierarchy. The workflow must be able to iterate certain branches, as during the productization phase, some instances might need customization of certain nodes, such as applying some different default values to all data attributes under a certain logical node. In the case of flattening, these operations would have to be done using a full $O(n)$ table scan that checks the string prefixes or building additional indexes, which would ultimately defeat the whole purpose of flat mapping the data. Another workaround would be to add each node point into the dictionary, which would work in theory, but would make the dictionary extremely large. The single dictionary would store each combination of logical devices, logical nodes, data objects, and data attributes.

Another approach would have been to implement a B-tree. Although a B-tree would keep the data in a sorted manner, this is not needed for SCL data and would introduce an unneeded balancing overhead. Although this would drop the search time complexity of the legacy tools $O(n)$ to $O(\log n)$. B-tree was also rejected as the solution due to its time complexity compared to the dictionary types $O(1)$ time complexity. This time complexity trade-off would be faced each time a new node is inserted into the tree, so roughly 350,000 for the current DUT 2 dataset.

The new data model is chosen to use a dictionary type from C#, which is an implementation of a hash table. The use of a dictionary type proved significant improvements in previous usage. Dictionaries are used to store values with type safety, preventing accidental insertion of incorrect node types. The dictionary class performs better than the “Hashtable” class because it avoids boxing and unboxing when storing or retrieving values (Microsoft, 2021).

The new data model uses a modified trie. Commonly trie is implemented with arrays of characters, but by replacing the arrays with dictionaries, it improves the lookup times greatly, as we are no longer bound by the size of n. Also, as noted in the technical

background for trie, it is implemented using single characters, whereas this modified trie uses IEC 61850 identifiers as the alphabet. The modified trie can be called a dictionary trie, or, more precisely, an N-ary tree with hash-map children. During this thesis, the data structure will be referred to as a dictionary trie.

Because the dictionary trie uses dictionaries rather than arrays, the implementation does not require a standard enumeration with string comparison. As seen in Figure 13, it implements the dictionary trie with IEC 61850, where each line represents a dictionary search operation within the model.

Searching the dictionary trie is dependent on the number of levels L, as each lookup is a dictionary lookup on each level, which are $O(1)$ on average, the average case to travel the L levels is $O(L)$. The performance of the dictionary trie is not dependent on the amount of data n that gets stored in the structure compared to the old structure.

Based on the nesting within existing ABB data, the maximum number of levels is 7, with 6,356 instances across 2,965 unique paths. The most common number of levels within the data is 4, with around 54,776 instances across 21,467 unique paths. The minimum number of levels is 3, with finished branches containing a data attribute with 3980 instances across 17 unique paths.

Given the current data, the tool's worst-case search operation will be 7 levels, and the best case only 1 level when fetching the logical device itself. Most common search operations will be done with 4 levels.

Although lookup performance depends on the number of levels in the data structure, the lookups are currently limited to at most 7 levels. However, the number of levels is subject to change, as more may be introduced later, though unlikely. Adding more layers might suggest some deeper structural problems within the SCL.

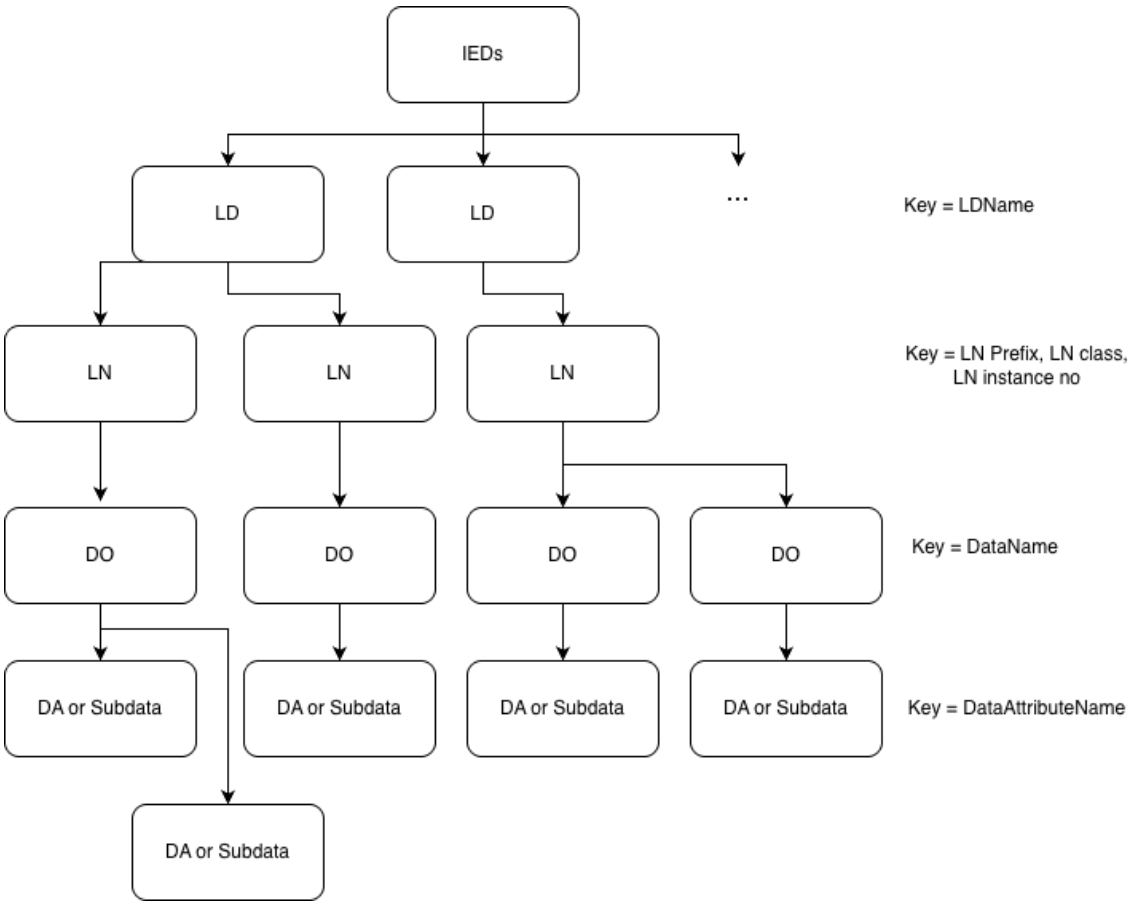

Figure 13. Implementation of a dictionary trie with IEC 61850 data.

Trie aligns well with the IEC 61850 data model hierarchy, as both are represented hierarchically, which makes it much easier to read multiple function designs in separate files and write data iteratively.

Using graph theory, the SCL hierarchy can be represented as a rooted tree where each node $v \in V$ contains a hash table H_v that stores the child nodes:

$$T = (V, E)$$

As SCL data makes some element attributes within the XML data mandatory, it's worth taking advantage of it by storing the node by using its elements of the signal identification as defined earlier in Figure 7 as the key for the hash-based data structure. Storing the data in a dictionary trie preserves the tree-like structure of the IEC 61850

data, which is beneficial when iterating through the data structure if the data must be written back to XML.

To insert an IEC 61850 node into the dictionary trie given a path such as LD0.UL1TVTR1.VolSv.sVC.scaleFactor, the model must check at each layer whether the node exists or should a new one be created. The insertion is much faster than the legacy implementation, as the data tree does not need to be walked through for the insertion, as mentioned in the performance analysis of the legacy implementation in the previous chapter.

The IEC 61850 data type template is also stored using the dictionary trie model. As IEC61850 makes it mandatory to have an ID attribute, it is selected as the key for accessing the value. However, for the data type templates, the search operation contains at most 2 levels and at least 1 level L. Accessing the type itself is a constant $O(1)$ operation and accessing the type members is a constant $O(2)$ operation.

Each type of data type template node is stored in a different dictionary, making it harder to accidentally assign an invalid type to a data node. Meaning a total of four separate dictionary tries are created for EnumType, DAType, DOType, and LNodeType.

As ABB has two private extensions, as shown in the SCL chapter. These private extensions lack uniqueness in their names as defined in their XML specifications. For the functionality extension, although no uniqueness is enforced, all existing components are created to be unique. The decision was to enforce uniqueness for current and future instances and modify the XML schema accordingly.

However, for ACT private extension, the uniqueness does not apply as it has multiple elements with the exact same name, with the exception that they have an interface identifier to separate them. The interface identifier only applies to ACT instances of hardware devices, such as modules within the IED, like X120 and X110, which are used

as the slot names for modules in the IED. There also exists a case where an ACT node also has a non-unique node name, but an identifying attribute. All these cases together invalidate the usage of a normal dictionary to store the data, so a multimap solution is chosen. For a single key, we return multiple instances as a list.

The new data model API also includes a read-only interface of the data model, which is useful as an input for parts where no modifications to the data model should be made. For example, the read-only model is used in code generation as its job is to only output code based on what the data model contains. Using the read-only model interface also adds an extra layer of protection to the model in case of an accidental modification attempt within the code.

To be able to generate the code, the types must be resolved for each data node. The algorithm (Figure 14) is a hierarchical type binding with recursive descent that traverses the SCL data. The algorithm starts by doing a search operation if a type is found for the element, if one is found it assigns the type for the data node. If no type is found, the resolver will exit immediately. As defined in the technical background, within each type, several members can exist, which must be mapped to the children of the current data node. This continues as a recursive operation until no children remain with unresolved types.

The algorithm is designed to be used as a single operation after all data has been added and must be explicitly called. The reasoning is that the tool is also used to edit instances of SCL, and to avoid unnecessary warnings during the design phase, the type templates may not exist at that moment.

The resolver itself can work in parallel by branching out a resolver for each logical device node within the data. This is possible due to each branch being an independent part within the tree, which allows concurrent operations on the data model branches.

Due to the recursive nature of the sub-data and data attribute nodes, the resolver utilizes a recursive function that has a stack protector to prevent the method from recursing too far, which uses a depth limit as a safety guard. The limit is chosen based on the maximum depth used in signal identification within ABB SCL models.

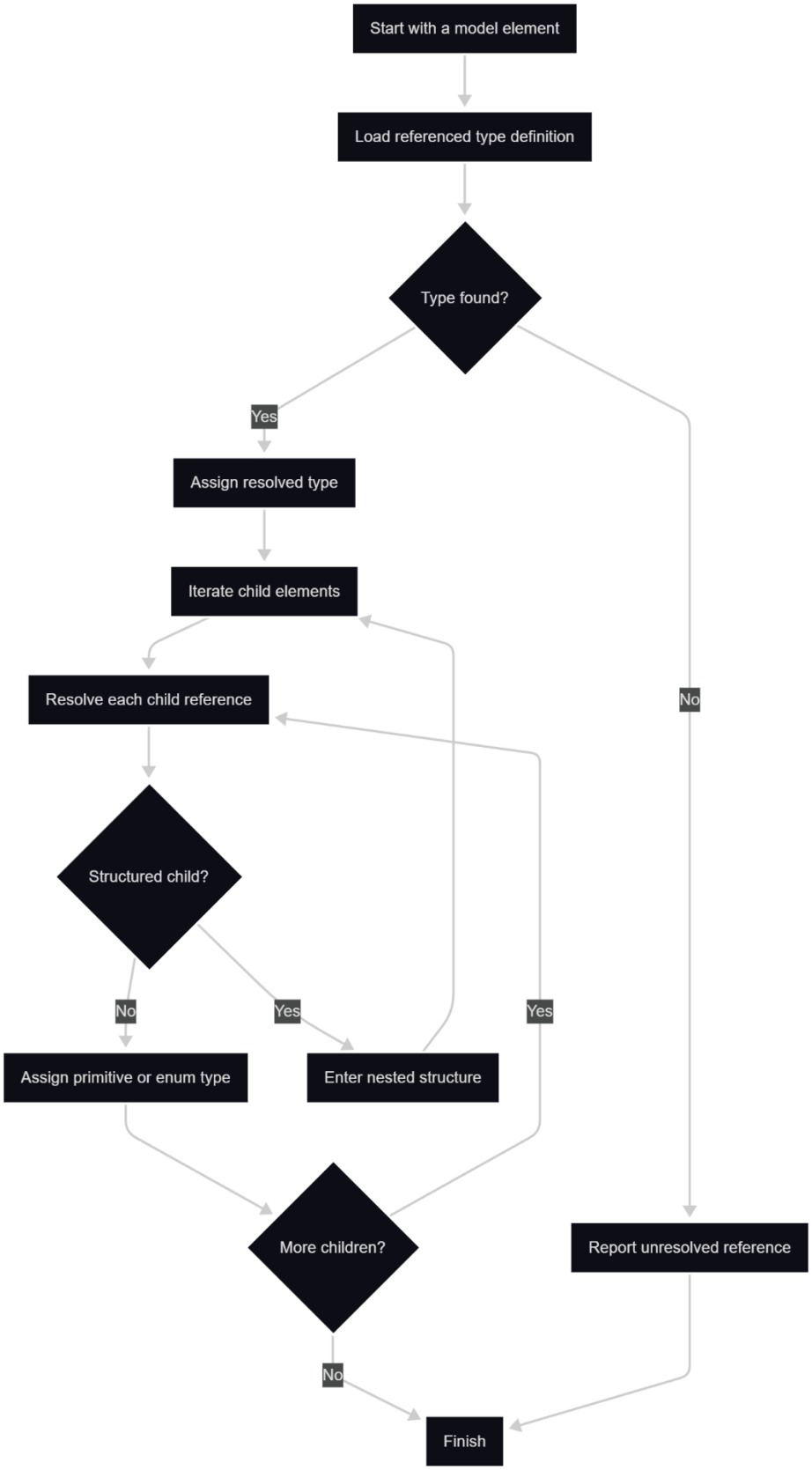

Figure 14. Flow chart of the type resolver algorithm.

5.2 Parsing Approach for SCL

To fill the model with the SCL data, it must be parsed. For working with XML data in C#, several choices exist. The XML data can be serialized and deserialized using the XmlSerializer class or read directly using a forward-only XmlReader or loading a complete DOM using XmlDocument (Microsoft, n.d.-a, n.d.-b, n.d.-c).

XmlSerializer uses dynamic code at runtime, which isn't viable as the application is published with NativeAOT. Building an application with a NativeAOT target and using XmlSerializer results in IL3050, which warns of this issue.

Usage of XmlDocument is also not seen as viable due to the huge size of the file, and XmlDocument loads the whole DOM into memory. In the research of (Jang et al., 2018) deem the usage of DOM-based solutions non-viable for SCL data.

The third choice is the XmlReader class, which provides a fast, noncached, forward-only access. XmlReader is chosen for the implementation based on the constraints of the problem with NativeAOT and the non-viability of DOM-based solutions with such large data (Microsoft, n.d.-c).

To create classes for each data type, XmlSchemaClassGenerator is used to map XML namespaces to C# namespaces (Ganss, 2013/2026). For IEC 61850 data, the IEC provided XML schema files are used, and for ABB private extensions, they are also generated from the XML schema files.

Using XML schemas to generate data offers several advantages, such as not having to manually write classes for each data type and having the benefit of easily modifying the schema classes, or in case of a new standard release, the migration can be done easily.

XmlSchemaClassGenerator does not, however, provide read or write functions and is designed to be used with the XmlSerializer. It marks classes with specific class and function attributes that the XmlSerializer then uses to make correct selections.

As reflection is also not viable due to the application being NativeAOT published the use of source generators is used as a workaround. The source generators generate reader and writer methods for all IEC 61850 data and ABB private extensions. Each class that a reader or a writer should be generated for is marked with a special attribute. The source generators work by inspecting user-created code that runs during compilation and creates additional code within the compiled product (Carter, 2020).

The created source generator generates an XML reader and XML writer methods for each class marked with a specific attribute. As XmlSchemaClassGenerator adds markers within the classes, its methods, and attributes, the source generator can take advantage of them when creating with the source generator.

The created reader methods take an instance of the XmlReader class, which is then used to get all element attributes under the current location. The writer methods, on the other hand, take an object instance that then writes all element attributes. With the use of XmlSchemaClassGenerator and source generator most of the SCL handling can be automatically created using the highly performant XmlReader class.

The parser is implemented as synchronous and asynchronous versions. The asynchronous implementation is added to support a GUI that uses the asynchronous version. The asynchronous implementation does not block the UI thread. The synchronous implementation is used in the CLI toolset, which generates the code and complete XML files, which are called from the terminal, where the blocking does not matter.

6 Evaluation and Performance Results

Performance evaluation is conducted using a benchmarking utility called BenchmarkDotNet v.0.15.2, and results are exported as HTML using the BenchmarkDotNet HtmlExporter.

All benchmarks are done in a sterile environment. Neither a debugger nor a profiler has been attached to the benchmark to avoid any performance overhead. The benchmarks are run with the “Release” configuration, which includes optimizations. As with the “Debug” configuration, Roslyn adds a lot of additional IL opcodes to the compiled result to help with debugging (Akinshin, 2019).

The benchmarks are run in an environment where all additional applications have been stopped. Only standard OS processes and the benchmark process are running. As natural noise and random errors can’t be prevented, multiple iterations of the benchmarks are done, and the distribution of the benchmarks is analyzed (Akinshin, 2019).

According to Akinshin (2019, pp. 520–521), in CPU-bound benchmarks, as the JIT compiler generates native code, its decisions can affect the performance. The compiler can decide to put variables on the stack or in registers. Registers usually work faster than the stack. The JIT can also decide to do inlining, which replaces the call to a method with its body, eliminating the call overhead, which can be disabled but cannot be forced, as inlining is not always possible. JIT also does instruction-level parallelism, which executes multiple instructions at the same time inside a single thread. Modern CPUs also perform branch prediction, which uses the history of taken branches across execution sessions. Intrinsics also come in both implicit and explicit versions. Implicit intrinsics usage depends on the current hardware, and it selects the best available one. Explicit intrinsics can be used manually to optimize using any hardware instructions.

Furthermore, Akinshin (2019, p. 571) describes that memory-bound benchmarks depend on the multiple levels of memory management happening between the CPU and the .NET runtime. The CPU cache contains the hot data and is placed inside the CPU, which requires very little time to access compared to accessing the main memory or accessing from disks. Memory layout is also important, which has a performance impact, such as unaligned memory, cache bank conflicts, and cache line splits, which are not expected by the benchmarks. As C# uses a garbage collector, it can affect benchmarks because it is nondeterministic, and overhead can be added at random.

Benchmarks are run on Windows 11 (10.0.22631.6783/23H2/2023Update/SunValley3) in High-Performance mode, with the laptop connected to an outlet. The laptop has an Intel Core Ultra 7 165H 1.40GHz CPU, 22 logical and 16 physical cores with 32GB of DDR5 RAM. The host is using .NET 10.0.5 (10.0.526.15411) with X64 RyuJIT AVX2. The benchmark results are specific to the hardware configuration mentioned, and other configurations may yield different values.

For each benchmark, BenchmarkDotNet generates an isolated project for each runtime setting and builds in Release mode. BenchmarkDotNet also handles iteration to calculate the overhead added by the tooling itself and uses the median of all overhead calculations to subtract from the actual workload measurements. BenchmarkDotNet will also iterate to warm up the JIT before the actual workload measurements are recorded.

6.1 Threats to Validity

The laptop has Microsoft Defender and other IT-managed antivirus software, which may cause occasional interruptions during benchmarking. As the device is managed, the antivirus can’t be set off during the benchmarks, nor can the source code be moved to other devices due to its proprietary nature. The benchmarks are executed multiple times, so any clear outliers caused by the environment are visible in the results. BenchmarkDotNet handles the outlier detection.

The first part of the benchmarking is conducted on the standard .NET RyuJIT runtime rather than the NativeAOT-compiled implementation, as the legacy implementation cannot be compiled for NativeAOT due to architectural constraints. This affects the absolute timing values, as the final implementation will use NativeAOT. Benchmarking the new implementation on NativeAOT while leaving the legacy implementation on JIT would result in an unfair comparison. Running both on JIT makes an apples-to-apples comparison, where the differences between data structures and algorithmic improvements are more clearly highlighted. A separate benchmarking was done with NativeAOT to measure the real impact of the final product.

Because the benchmarks will use synthetic data, the data may not fully reflect the structural characteristics of real SCL data. The synthetic data may affect hash distribution and cache locality. Therefore, results using synthetic data should be examined as stress tests that demonstrate scalability rather than as performance predictions for real future datasets. The synthetic generates data at all levels of the SCL, also creating thousands of logical devices that do not represent a realistic case of a single IED, as data distribution within existing configurations has at most 5 logical devices. Other parts of the data distribution, such as logical nodes, data objects, and data attributes, do represent a somewhat realistic distribution.

6.2 Benchmarking

The data model must be high-performing for large datasets. To benchmark the application's performance, benchmarks have been created for the data model, parsing approach, code generation throughput, and SCL throughput.

The data model benchmarks focus on search operations that traverse the dictionary trie through its layers, as well as insert and deletion operations throughout the dictionary trie. The model iteration is benchmarked at multiple levels, including all logical devices, logical nodes, data objects, and data attributes.

The data type resolver is also benchmarked on how it can handle the data and how parallelization affects the results compared to a data type resolver without parallelization. The previous tool versions had their data types resolved on import, which required the import of the data type templates first. Meaning the previous versions could not handle the resolver concurrently.

Comparisons are made of the new implementations against legacy targets with the dictionary added, reflecting the cumulative effect of all improvements combined. Comparing these two legacy targets against the new implementation provides valuable data on why a complete rewrite of the software was warranted and shows the importance of using data structures that fit within the constraints of the problem domain.

6.2.1 SCL Dictionary Trie Operation Benchmark

To test the performance of search operations, the dictionary trie data model is loaded with the DUT 2 data model. The benchmark will test lookups of the model of the same data nodes that were tested earlier in the performance analysis of the legacy model. Insertion and deletion operations are performed on an empty data model.

Lookups from the dictionary trie depend on the count of layers L in the trie as the search goes deeper, a new lookup must be done. The benchmark results also reflect this, as searching for a logical device costs 9.336 ns. Traversing deeper and doing a lookup of a logical node within the device, the cost increases to 14.146 ns total. The trend continues with doing lookups within the further levels, as depth increases, the lookup time increases. With the .NET 10.0 runtime, the mean time for all operations is under 50 ns within a depth of 7 (Figure 15, Table 3).

To reflect on the earlier data of the legacy implementation, a last node search is also performed. As can be seen, the position of the node within the same layer in the data structure does not affect the lookup time negatively.

Performing a search at the same depth 4 with the new data model compared to the legacy implementation, 335,909.1 ns, the lookup factor is improved by 15,601.2x. The 335,909.1 ns is also the worst-case scenario for the old data model, with the new data model, the worst-case scenario is a depth 7 search, which takes at most 47.595 ns, improving the lookup by 7057.7x.

Performing a lookup within the data model has the benefit of doing zero allocations. Compared to the legacy, which required allocations for both the improved implementation and the prior solution.

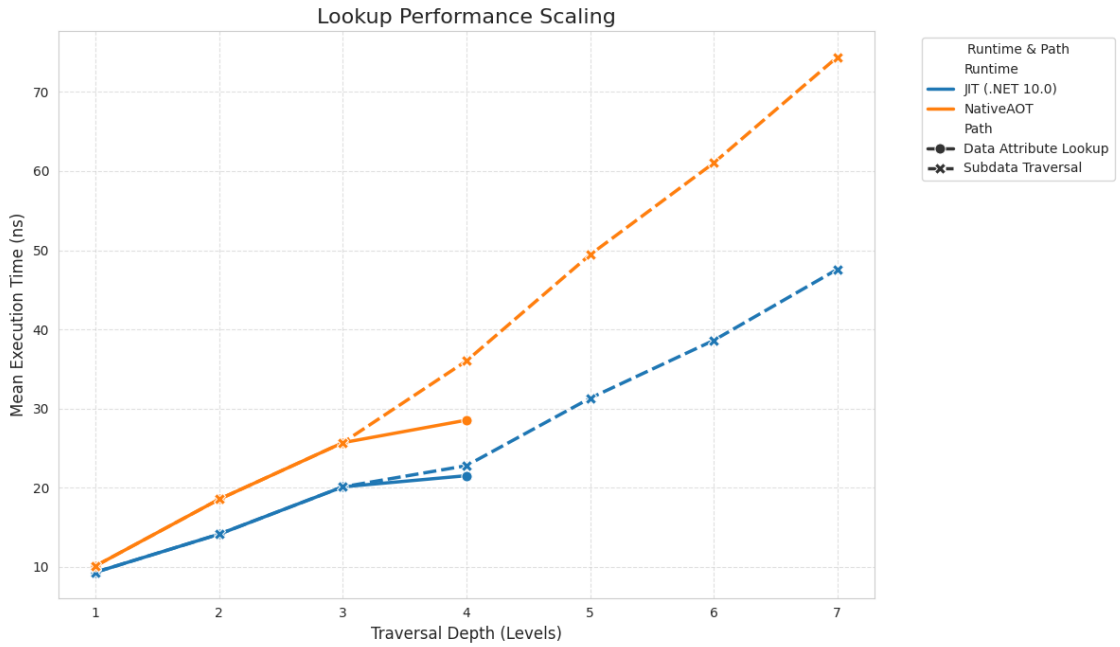

Figure 15. Lookup performance scaling with traversal depth using .NET 10.0 and NativeAOT 10.0 runtime.

Running the same benchmarks with NativeAOT 10.0 at depth 1 of finding the device node takes a mean time of 12.213 ns. Going further and fetching a logical node within a device takes a mean time of 22.611 ns with an increase of 85%. Fetching a data object from a logical node takes 32.658 ns, an increase of 44%, and fetching a data attribute takes 44.572 ns, an increase of 36%.

The performance of search operations in the NativeAOT 10.0 runtime is slightly slower than in the .NET 10.0 runtime. The performance of .NET 10.0 runtime is better due to the dynamic profile-guided optimization (PGO) feature of the JIT compiler (.NET Platform, 2019/2026). The dynamic PGO works by monitoring benchmark execution and, when it identifies a hot path, recompiles the method using tier-1 compilation. As the JIT knows the types at runtime, it can devirtualize interface calls, inline methods, and strip away safety checks. The NativeAOT version is compiled ahead of time and has no JIT, and cannot do the same kind of dynamic PGO as the .NET 10.0 runtime.

The optimizations that have been made can be verified by disassembling the benchmark, which is dumped after the benchmark is run. The disassembly verifies that the PGO has used guarded fast paths and hot-path-first-layout.

Running the lookup benchmarks without the tiered PGO, the lookups are now quite similar to the NativeAOT runtime benchmarks (Figure 16). The comparison verifies that the PGO does optimizations that the NativeAOT is not capable of performing, and which benefit the NET runtime.

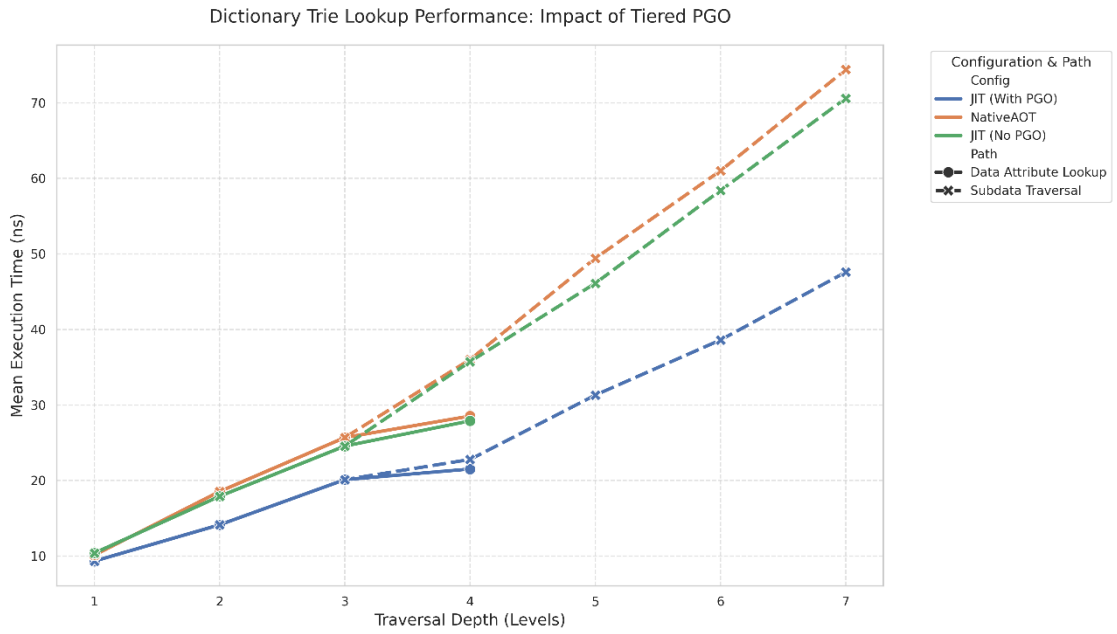

Figure 16. Impact of tiered PGO on lookup performance with traversal depth using .NET 10.0 runtime and NativeAOT 10.0.

NativeAOT also does generic code sharing called canonicalization, which is used to keep the app small by calling the same machine code for multiple Dictionary types using a string, object, as well as string, string, and so on. The canonicalization usage can be directly verified by viewing the disassembly output of the benchmark. As seen in Figure 17, the call operation on row 23 can be seen calling the dictionary with "__Canon" arguments, verifying the usage of canonicalization within the NativeAOT runtime.

```
15 G_M000_IG02:
16       mov      r8, gword ptr [rcx+0x08]
17       mov      rcx, gword ptr [r8+0x30]
18       test     rcx, rcx
19       je       G_M000_IG10
20       lea      r8, [rsp+0x20]
21       mov      r11, 0xD1FFAB1E
22       mov      rdx, 0xD1FFAB1E
23       call     [r11]System.Collections.Generic.IReadOnlyDictionary`2[System.__Canon,System.__Canon]:TryGetValue(System.__Canon,byref):bool:this
24       xor      rcx, rcx
25       test     eax, eax
26       cmovne   rcx, gword ptr [rsp+0x20]
27       xor      rdx, rdx
28       mov      gword ptr [rsp+0x20], rdx
29       test     rcx, rcx
30       je       SHORT G_M000_IG08
```

Figure 17. Disassembly of NativeAOT benchmark showing canonicalization.

Insertion into the data model (Table 5) also depends on the level of the insert, as each insert must look up the previous level where it should be added. If no such level exists, it is created in the data model. All insertion operations that are done from depth 1 to depth 7 are below 8500 ns. For the insertion operations, the prior levels were created before insertion into the dictionary trie, so the benchmarks directly measure the combination of search operations up to the desired layer and the insertion into that layer.

Deletion from the data model (Table 4) also follows the same principles of requiring a lookup to find the correct data node to delete from the model. For the deletion benchmarks, the data is created in the data model prior to starting the benchmarks to purely measure the combination of the search operations and the deletion operation at the desired level.

Comparing the results of the .NET 10.0 and NativeAOT 10.0 runtimes for insertion and deletion operations, the NativeAOT 10.0 runtime performs better than the .NET 10.0 runtime (Figure 18). Only the data attribute insertion is slower in the NativeAOT runtime compared to the .NET runtime. The largest improvements are within deletion operations with third-level sub data.

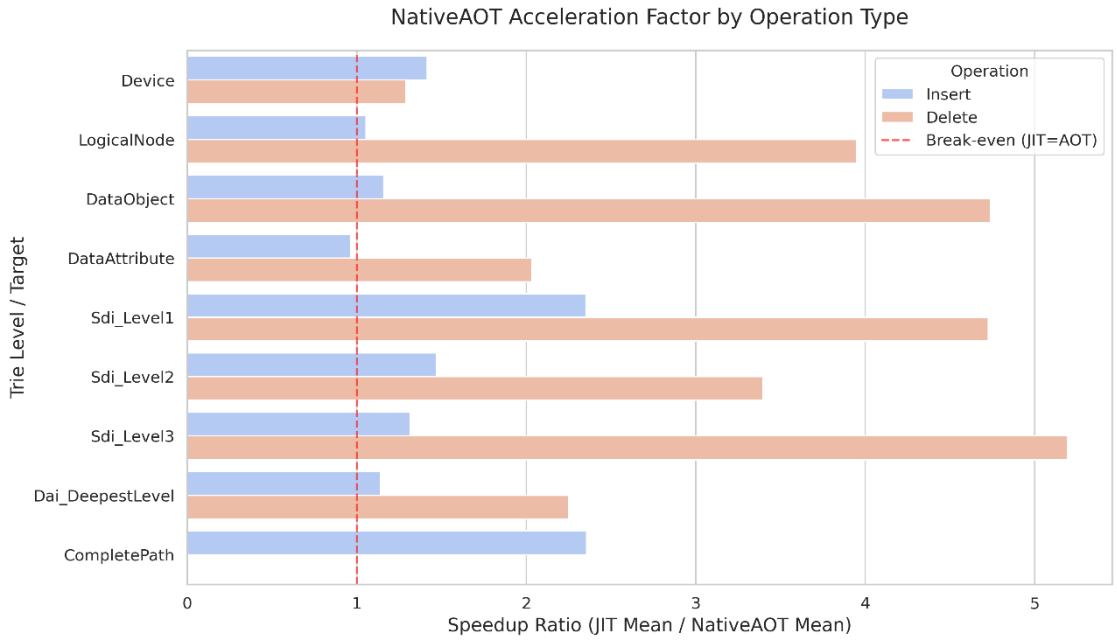

Figure 18. NativeAOT acceleration factor by operation type.

Table 3. Search operation times with dictionary trie for .NET 10.0 and NativeAOT 10.0 runtime.

Method	Runtime	Mean (ns)	Error (ns)	StdDev (ns)	Depth
FindDevice_ByName	.NET 10.0	9.336	0.5494	1.6112	1
FindLn_ByName	.NET 10.0	14.146	0.3532	0.6889	2
FindDo_ByName	.NET 10.0	20.122	0.9256	2.7291	3
FindDaiRef_ByName	.NET 10.0	21.531	0.7102	2.0829	4
FindDaiRef_ByName_LastNode	.NET 10.0	24.191	1.8145	5.2354	4
GetSdi_Level1	.NET 10.0	22.795	0.5077	0.7441	4
GetSdi_Level2	.NET 10.0	31.317	0.6748	0.9237	5
GetSdi_Level3	.NET 10.0	38.618	0.8197	1.1756	6
GetDai_DeepestLevel	.NET 10.0	47.595	0.9890	1.4183	7
FindDevice_ByName	NativeAOT 10.0	10.133	0.2587	0.3627	1

FindLn_ByName	NativeAOT 10.0	18.552	0.4313	1.1134	2
FindDo_ByName	NativeAOT 10.0	25.698	0.5441	0.9239	3
FindDaiRef_ByName	NativeAOT 10.0	28.544	0.6171	1.7099	4
FindDaiRef_ByName_ LastNode	NativeAOT 10.0	42.959	2.3361	6.8880	4
GetSdi_Level1	NativeAOT 10.0	36.024	0.7709	0.6834	4
GetSdi_Level2	NativeAOT 10.0	49.436	1.0509	1.7844	5
GetSdi_Level3	NativeAOT 10.0	61.024	1.2852	2.5368	6
GetDai_DeepestLevel	NativeAOT 10.0	74.405	1.5483	4.1594	7

Table 4. Data model deletion times for .NET 10.0 and NativeAOT 10.0 runtime.

Method	**Runtime**	**Mean (ns)**	**Error (ns)**	**StdDev (ns)**	**Median (ns)**	**Depth**
Delete_Device	.NET 10.0	1,724.5	357.21	1,019.1	1,500.0	1
Delete_LogicalNode	.NET 10.0	4,329.2	704.91	2,033.8	4,100.0	2
Delete_DataObject	.NET 10.0	4,728.9	773.05	2,242.8	4,200.0	3
Delete_DataAttribute	.NET 10.0	5,036.2	803.37	2,292.1	4,250.0	4

Delete_Sdi_Level1	.NET 10.0	5,717.2	866.75	2,458.8	5,100.0	4
Delete_Sdi_Level2	.NET 10.0	6,789.8	1,093.07	3,188.5	5,450.0	5
Delete_Sdi_Level3	.NET 10.0	4,168.0	498.82	1,447.2	3,800.0	6
Delete_Dai_DeepestLevel	.NET 10.0	6,245.7	934.26	2,665.5	5,750.0	7
Delete_Device	NativeAOT 10.0	1,336.4	101.29	297.1	1,300.0	1
Delete_LogicalNode	NativeAOT 10.0	1,096.8	93.04	263.9	1,000.0	2
Delete_DataObject	NativeAOT 10.0	997.9	114.59	326.9	900.0	3
Delete_DataAttribute	NativeAOT 10.0	2,477.3	204.74	594.0	2,600.0	4
Delete_Sdi_Level1	NativeAOT 10.0	1,210.0	151.25	446.0	1,100.0	4
Delete_Sdi_Level2	NativeAOT 10.0	1,999.0	225.71	658.4	1,900.0	5
Delete_Sdi_Level3	NativeAOT 10.0	802.2	71.14	197.1	800.0	6
Delete_Dai_DeepestLevel	NativeAOT 10.0	2,776.0	298.02	878.7	2,650.0	7

Table 5. Insertion times for .NET 10.0 and NativeAOT 10.0 runtime.

Method	Runtime	Mean (ns)	Error (ns)	StdDev (ns)	Median (ns)	Allocated (B)	Depth
Insert_Device	.NET 10.0	6,155.3	451.96	1,289.5	6,100.0	656	1
Insert_LogicalNode	.NET 10.0	5,169.5	493.82	1,416.8	4,700.0	808	2
Insert_DataObject	.NET 10.0	5,565.6	430.31	1,241.5	5,400.0	1144	3
Insert_DataAttribute	.NET 10.0	4,442.6	313.14	893.4	4,300.0	336	4
Insert_Sdi_Level1	.NET 10.0	5,541.8	382.18	1,071.7	5,300.0	752	4
Insert_Sdi_Level2	.NET 10.0	5,824.1	303.54	830.9	5,600.0	752	5
Insert_Sdi_Level3	.NET 10.0	8,415.5	527.37	1,530.0	8,400.0	752	6
Insert_Dai_DeepestLevel	.NET 10.0	6,164.9	444.24	1,267.4	5,900.0	472	7
Insert_CompletePath	.NET 10.0	17,483.0	2,548.72	7,271.6	15,200.0	5080	7
Insert_Device	NativeAOT 10.0	4,348.4	381.52	1,094.7	4,200.0	656	1
Insert_LogicalNode	NativeAOT 10.0	4,898.9	521.69	1,471.4	4,500.0	808	2
Insert_DataObject	NativeAOT 10.0	4,802.1	389.76	1,112.0	4,500.0	1144	3
Insert_DataAttribute	NativeAOT 10.0	4,614.7	420.21	1,205.7	4,300.0	336	4

Insert_Sdi_Level1	NativeAOT 10.0	2,355.1	169.62	470.0	2,300.0	1040	4
Insert_Sdi_Level2	NativeAOT 10.0	3,956.6	338.91	994.0	3,600.0	1376	5
Insert_Sdi_Level3	NativeAOT 10.0	6,390.0	655.81	1,933.7	5,950.0	752	6
Insert_Dai_DeepestLevel	NativeAOT 10.0	5,406.6	427.24	1,198.0	5,200.0	472	7
Insert_CompletePath	NativeAOT 10.0	7,421.1	468.24	1,305.3	7,300.0	5368	7

6.2.2 SCL Parser Benchmark

To benchmark the new improved model, the benchmarks use real data from the previously mentioned DUT 1 and DUT 2 data models, which were also used to test the legacy implementation performance and to prove its scalability for the future. The benchmark also tests using synthetic data models, which are 2x and 5x the size of the DUT 2 model. The performance of the parser is heavily dependent on the insertion speeds of the data model.

As mentioned in an earlier chapter, the DUT 1 model contains 5 logical devices, 1141 logical nodes, 18811 data objects, and 212846 data attributes, resulting in a total of 232803 data nodes across 567 files.

The synthetic data is generated using an additional script that generates randomly named functions with no actual functionality.

The parser allows the usage of an asynchronous version and a synchronous version. The synchronous version performs a bit better than the asynchronous version. The load time for all nodes using the synchronous approach is 498.636 ms, whereas the asynchronous approach performs at 623.019 ms.

With the assumption that the files are equivalent in size at 623.019 ms for the full 567 files, each file loads at 1.0987 ms, whereas for the synchronized version, at 498.636 ms for the full 567 files, each file loads at 0.8794 ms. The synchronous version performs 25% better than the asynchronous version. This is explained by the additional overhead that the asynchronous versions add.

The synchronous version also allocates 229190.28 KB compared to the 300342.63 KB of the asynchronous version. The additional allocations are also the result of the asynchronous overhead of allocating additional locks for the data structure.

Table 6. Performance for improved parser.

Method	**Mean (ms)**	**Error (ms)**	**StdDev (ms)**	**Gen0**	**Gen1**	**Gen 2**	**Allocated (KB)**
LoadAllDUT1Nodes_Sync	498.636	5.2226	4.0775	21000	17000	3000	229190.28
LoadAllDUT1Nodes_Async	623.019	9.4161	8.3471	25000	15000	1000	300342.63

The contribution is much more significant when compared to the legacy implementations. Testing the application's performance without the added dictionary, parsing the parser takes 10 minutes to build the SCL data model. Even after the implementation of the dictionary caching layer, the legacy implementation takes a long time to collect all 567 XML instances. The parser of the legacy implementation takes 152220 ms to load all the instances, which translates to a bit over 2.537 minutes (Table 7), with an error of 516ms and a standard deviation of 480 ms.

The new implementation provides much better performance than the previous, with a performance improvement of 99,67% or a 305.3x speedup factor, meaning the new

implementation is 305.3 times faster than the legacy implementation when compared to the synchronous implementation.

Table 7. Legacy parser performance for DUT 1 dataset.

Method	Mean (ms)	Error (ms)	StdDev (ms)	Gen0	Gen1	Gen2	Allocated (GB)
LoadAllDUT1Nodes Legacy	152220	516	480	282000	38000	4000	3.26

Loading the DUT 2 model, which contains 5 logical devices, 1589 logical nodes, 28102 data objects, and 320951 data attributes across 769 files. To load the complete DUT 2 model with the synchronous parser, it takes a total of 854.5 ms. Compared to the DUT 1 model which had a total of 232803 data nodes which were loaded at 498.636ms the DUT 2 loads a total of 350647 data nodes at 854.5ms. Loading 117844 nodes costs 355.864ms more.

The DUT 1 model is loaded at 0.0021419 data nodes per millisecond, whereas the DUT 2 is loaded at 0.0017768 data nodes per millisecond. As the data nodes per millisecond does not increase when adding more data.

Table 8. Improved load time for DUT 2 dataset.

Method	Mean (ms)	Error (ms)	StdDev (ms)	Gen 0	Gen 1	Gen 2	Allocated (MB)
LoadAllDUT2Nodes Sync	854.5	17.06	16.75	33000	28000	4000	351.92

Comparing the new parser with the legacy parser with an intermediary dictionary applied. The legacy implementation takes 315360ms (5.256 minutes) to parse all data nodes from the 769 files while allocating 6.21GB of memory.

The speedup factor between the legacy implementation and the new implementation is 99.73% or a 369.1x. As the amount of data n rises, the speedup factor continues to grow, favoring the new solution due to the new implementation being only dependent on the number of levels L.

Table 9. Legacy load time for DUT 2 dataset.

Method	**Mean (m)**	**Error (m)**	**StdDev (m)**	**Gen 0**	**Gen 1**	**Gen 2**	**Allocated (GB)**
LoadAllDUT2Nodes_Legacy	5.256	0.0329	0.0307	537000	63000	6000	6.21

When comparing the results that have been gathered, the speedup factor keeps rising as data keeps getting bigger. This proves that the new model and parser implementation to be much more performant and scalable than the legacy implementation.

Testing with the generated data by doubling the number of files to read to 1538. The generated dataset contains a total of 1538 logical devices, 5362 logical nodes, 30919 data objects, and 102025 data attributes. This increases the load time by 262.7 ms, from 854.5 ms to 1117.2 ms. Raising the number of generated files to 5x of DUT 2, a total of 3845 files. Generated files contain 6152 logical devices, 21550 logical nodes, 124549 data objects, and 408631 data attributes. Increasing the parsing time by 868.8ms from 1117.2ms to 1986.0ms. Finally, the model is tested on a 10x of DUT 2 with a total of 7690 files containing a total of 13842 logical devices, 48441 logical nodes, 279851 data objects, and 917879 data attributes. Going to 3500.1ms from 1986.0ms shows an increase of 1514.1 ms (Table 10).

With each dataset, the mean time spent rises in a linear fashion as seen in Figure 19, which is made from the data shown in Table 11. This result makes sense as most work is done going through the files linearly. As the parser approaches a new data node, the

search operation is done on the current level, which is a constant-time operation. Parser then either creates a data node on that level or uses the existing node found. New children are then inserted into the previously found node, which again is a constant-time operation.

Table 10. Mean parsing time in each dataset.

Configuration	Mean (ms)	Error (ms)	StdDev (ms)	Gen0	Gen1	Gen2	Allocated (MB)
DUT2x2	1117.2	17.69	15.68	43000	38000	4000	473.14
DUT2x5	1986.0	31.95	28.33	74000	69000	5000	837.47
DUT2x10	3500.1	54.59	48.40	126000	119000	6000	1439.4

Table 11. Data metrics of tested configurations.

Configuration	File Count	Logical Devices	Logical Nodes	Data Objects	Data Attributes
DUT2	769	5	1589	28102	320951
DUT2 x2	1538	1543	6951	59021	422976
DUT2 x5	3845	6157	23139	152651	729582
DUT2 x10	7690	13847	50030	307953	1238830

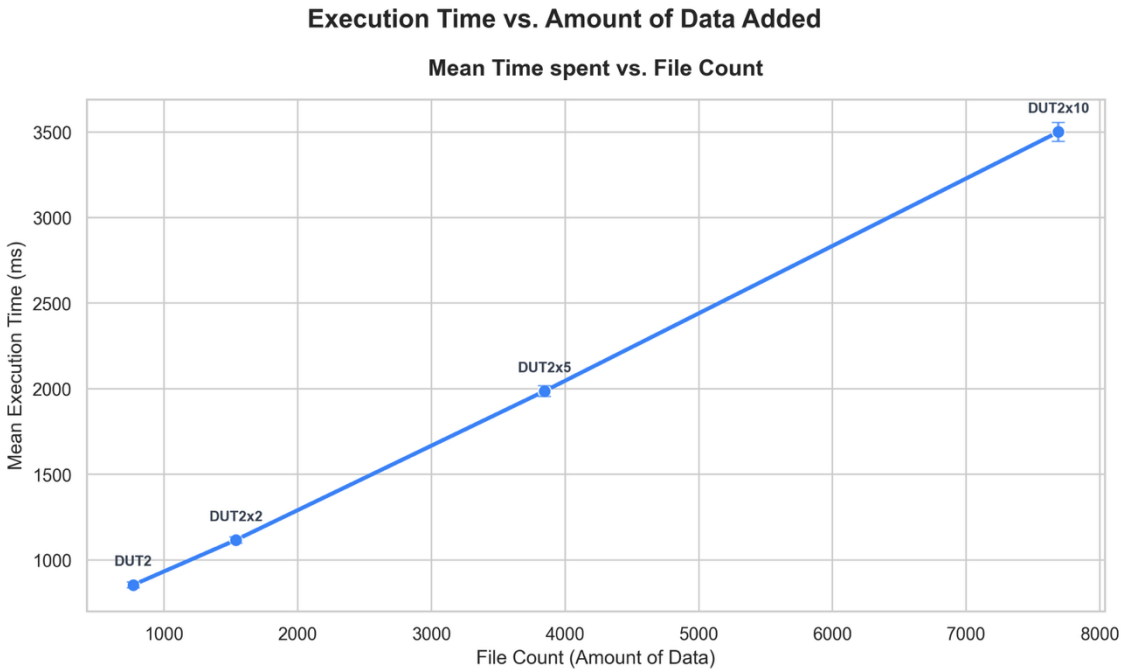

Figure 19. Mean time spent chart against amount of data added.

6.2.3 SCL Complete Workflow Benchmark

To test the capability of the tool's main purpose of code generation and the compilation of the final complete XML benchmarks are defined for those workflows. The first benchmark tests the throughput code generation of a single function design. The second benchmark takes the same DUT 2 and DUT 1 datasets as introduced previously and does a throughput test of reading in all the files and then constructing the complete model. As in previous benchmarks, these are compared to the legacy implementation with the intermediary dictionary applied.

Testing the implementation of code generation throughput with all function designs with varying data sizes. The results give a range of 195.80 µs to 162720.10 µs and a mean of 1656.62 µs for C code generation. As more data is added, the longer the generation will take as every node must be visited making it an $O(n)$ operation. Pearson correlation analysis was performed between the amount of the elements and the execution time, resulting in a correlation coefficient of 0.69. The correlation coefficient shows that the

specific types of elements and their depth introduce variance independent of the file size.

The 162,720.10 µs is the SMVSENDER IEC61869-9 function, which can be seen as a clear outlier in the data (Figure 20). Every other function manages to get a generation time of under 20,000 µs. One of the main reasons why SMVSENDER IEC61869-9 is much slower than other instances is its amount of data, as shown in Figure 21. It is the most complex function design with 8799 DAI instances alone. However, reflecting the data with the processing time within Figure 22 shows that the datatype resolution or the C code generation is unusually expensive, shown in spikes within the figure.

The statistics under Figure 20 does not include all instances, as some are missing data types within the SCL data, and fixing the source SCL files themselves is out of scope. The code generation can generate 506 function designs out of the 769 function designs, and other instances fail due to missing data types in the SCL files themselves, which represents the incompleteness of the input rather than failure within the tool logic. The failing instances have been filtered out of the benchmark to represent a clean benchmark of the tool's capabilities.

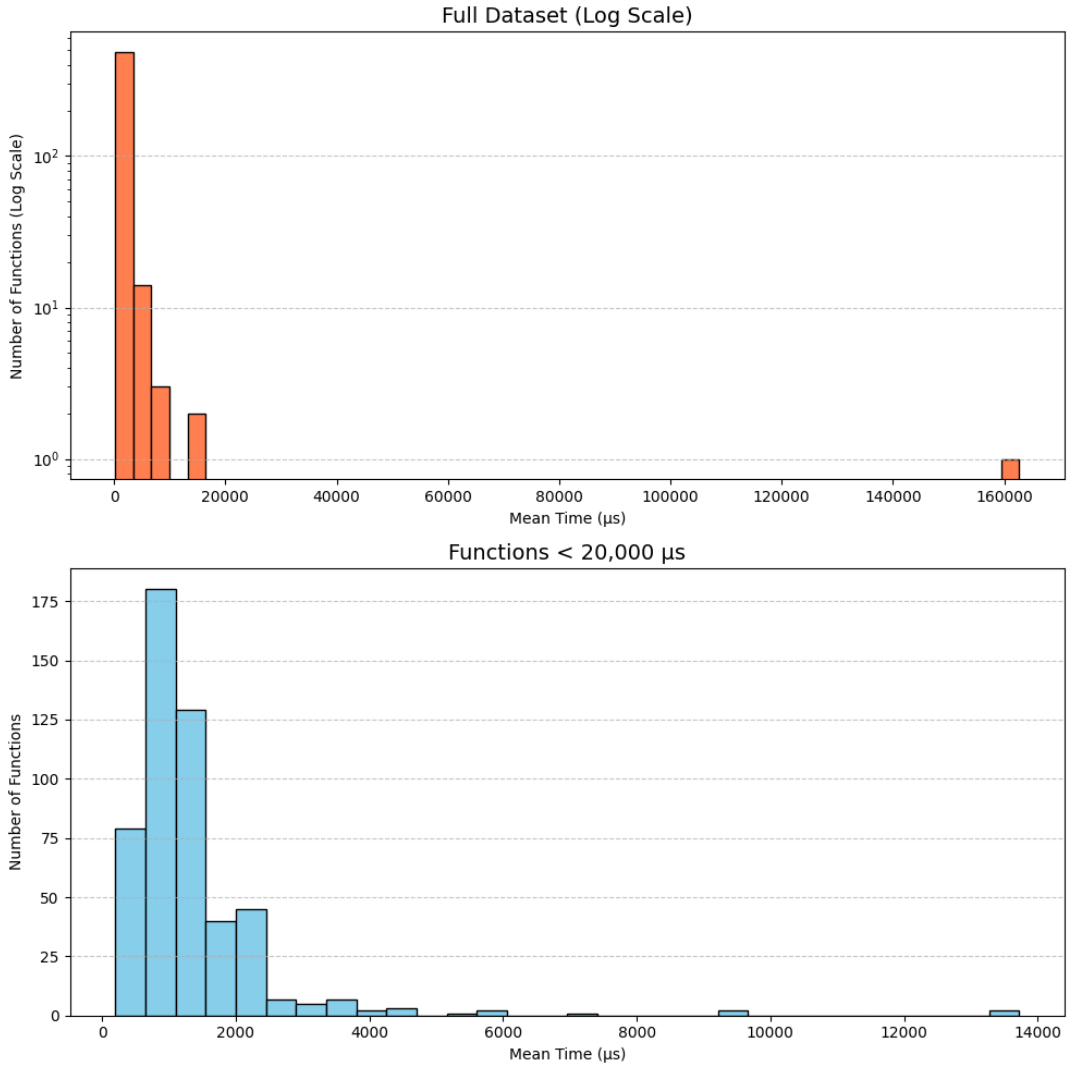

Figure 20. Function design C code generation time for all instances.

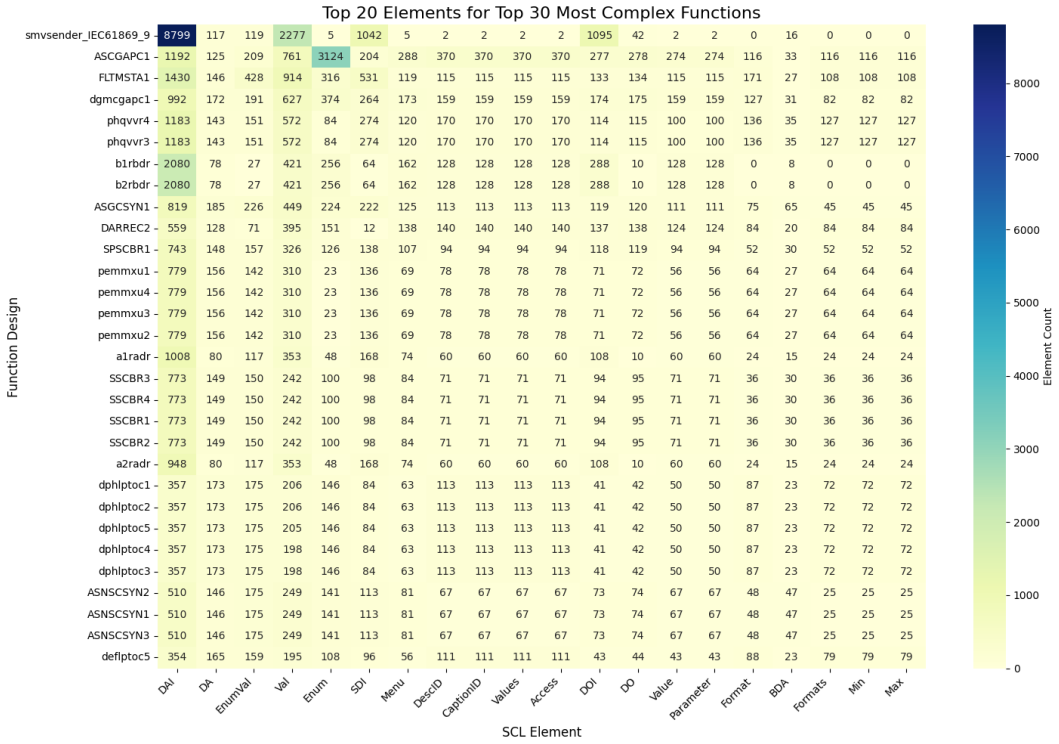

Figure 21. Elements for the most complex function designs.

Figure 22. Processing efficiency for C code generation of function designs.

The benchmark of reading all the files and writing all the files into a single combined file for the DUT 1 dataset takes a total of 2.144 seconds. For DUT 2, the complete read and write into a single file takes 4.290 seconds. Showing an expected increase in the execution time when more data is handled. Given the reading and writing also being $O(n)$ process the result makes sense.

Table 12. Complete SCL file write times for DUTs.

Method	Mean (s)	Error (s)	StdDev (s)	Median (s)
Complete_Writer_DUT1	2.144	0.0416	0.0369	2.160
Complete_Writer_DUT2	4.290	0.0975	0.2843	4.184

The code generation, along with writing the complete XML file for the larger device, can easily meet the target execution time of under 5 minutes at roughly 10 seconds for the complete workflow of DUT 2.

6.3 Impact of Implementation

The impact of this new implementation is quite large as it saves development time in the day-to-day operations and benefits the CI/CD pipelines with lower execution times.
The new execution pipeline approach also benefits by reducing the file size and gives a much cleaner syntax to work with, which reduces the complexity at the same time. The syntax can also be supported with a schema, giving the developers error reporting before the tool itself is called by utilizing a language server protocol (LSP) to give warnings visible in the user's editor of choice.

The new tool implementation has cumulative benefits due to its massive time savings. Given a standard execution time of 2h 40min for a DUT 2 configuration, run as a nightly build. The new implementation can do the work in a few seconds, such as writing the complete XML and generating code for each function design. Cumulatively for the year, assuming no build failures and the dataset has no additions, and assuming a total workflow takes at maximum 1 minute, then 58,035 minutes are saved each year for DUT 2 configuration. As ABB has two major devices, the migration of these devices to the new tool saves a large amount of time.

The time savings are directly noticeable by developers doing day-to-day activities with the tool. Previously, build times were so long that developers jumped to other tasks while the build was running, whereas this minimizes the chance of the developer losing focus on the task at hand. Assuming even a single developer uses the tool daily and builds function design X and then builds the configuration for device Y once a day, this calculates at approximately 150 minutes for each developer daily, calculating tool usage with a standard 5-day work week, assuming around 250 working days a year, each developer is looking at saving 37,500 minutes each year. The performance of this work also proves the viability of using dictionary tries for IEC 61850-6 SCL data with the constraints defined by this work.

As ABB does test automation for each build. Testing can be triggered earlier than previously possible due to the amount of time saved. Often during daily meetings, which are held in the morning, the test automation could still be running when we should be discussing its results.

7 Conclusions

The new, more scalable tool benefits ABB greatly by saving time from the C code generation and SCL data generation. As ABB now releases new firmware versions more often, the new tool version removes much of the previous tool's burden, such as taking a significant portion of the build time.

The new codebase was developed with a modern architecture and clear separation, making it more maintainable for others in the future. The new architecture allows for easy swapping of components with new implementations and requirements. The components are also heavily unit-tested, whereas the previous ones weren’t and lacked unit tests entirely. The clear separation also allows developers to tackle problems much more easily than with single-responsibility classes, which were tangled with many other classes.

Tracing proved to be a valuable tool for optimizing both the legacy tool and the new code. Through tracing, several hotspots were identified as the primary causes of the poor execution times, which ultimately warranted the thesis.

To implement the new tool, rigorous asymptotic analysis was used to design the dictionary trie and multimaps, as well as the algorithms around them. Benchmarking around real SCL data and synthetic data confirmed the implementation's performance and that it matches the initial analysis of the design phase. Due to benchmarking, the solution could be compared across different runtimes, revealing differences in search, insertion, and deletion operations within the data model. The gathered data enables informed decisions and further optimization.

The tool handles SCL data at high speeds and is highly scalable. As shown in the results, the tool can easily handle IED data models that are 2x, 5x, and 10x the size of the DUT 2 dataset. The speedups of 305x for DUT 1 and 369x for DUT 2 are very successful, even

when compared to previous improvements that included the intermediate hash map implementation. The lookup times improved significantly with the new data model, yielding a 14,165.6x speedup at a matching depth of 4 compared to the legacy implementation.

The solution can be used well into the future, given current performance and accounting for hardware improvements, and, as of now, it already handles the larger loads demonstrated in the benchmarking section. As measured in the complete workflow benchmarks, the solution can target an under-5-minute build time. The complete workflow takes under 10 seconds, which will be further improved once implemented in the build environment, which parallelizes tasks. The overall workflow is 960x faster and reduces execution time by 99.896% compared to the legacy solution.

Given the complexity of IEC 61850-6 SCL data, the format is the bottleneck on the performance of other applications as well. Larger substations can have much more data than what was tested with this SCL solution. These substations can feature multiple IEDs, counted in tens, that output large amounts of SCL data. This solution could replace the data model within slow configuration tools if the remaining features were implemented.

Other than configuration tools, the dictionary trie is well-suited for use with data other than SCL. The data requires preserving the hierarchy, is keyed, and must be mutable. If none of those constraints fit the processed data, the dictionary trie might not be fit for the problem.

7.1 Lessons Learned

The scope of the project was quite large as the tool contains 20 years' worth of features, and the schedule was tight. Modernizing a legacy monolith and achieving a complete feature parity within 5 months is a tight fit therefore the tool can't meet total feature parity, and it must be continued after the thesis. The thesis prioritized the core

implementation with the high-value workflows to build a good foundation that can be expanded upon.

Although the previous codebase worked fine, it desperately needed a rewrite to improve maintainability and clear separation of responsibilities. The architectural anti-patterns slow the development of new implementations such as retrofitting unit tests or swapping out data structures. This was experienced first-hand through earlier attempts to modernize the codebase.

The importance and usefulness of profiling applications are extremely beneficial, enabling the identification of hotspots in real-world usage. The benchmarking works great for verifying performance claims. The usage of these tools protected the implementation from introducing unnecessary complexity or trying to do any wrong optimizations.

7.2 Future Work

Future work will expand the UI capabilities beyond the current implementation, which provides only the bare-bones UI and does not meet all the requirements for a total replacement. The work will also focus on onboarding the tool to replace the usage of the previous tool, such as converting all previous files used for scripting purposes to the new YAML-based format. The conversion is done only for new development, and earlier products will continue using the legacy solution as they are not actively developed and therefore changing the tool would carry an unnecessary risk. The legacy solution will be kept in use in old devices until they are no longer maintained.

The mapping file format could also be improved to another format, such as a YAML-based one, which would reduce file size and make it much more human-readable and declarative. The mapping file format is overtly complex for its simple use case.

The tool was migrated to .NET 10, a long-term support (LTS) release. Future iterations will follow the release cycle of .NET LTS releases. Each LTS release is made in even-numbered years, meaning a major update is released every other year. Patch releases of .NET are strictly followed and applied to the tool.

The ABB private extensions are very difficult to parse and takes up extensive space in each SCL file. To improve the situation, some object tags could directly omit some elements from the object attributes. This would drastically reduce the size of the objects and reduce the additional work needed to parse them.

Within the commonSA namespace, the Object element contains a Name and a Node element, which could be easily combined into a single Object element. This alone would reduce the number of rows by 2 for each object. This is also noticeable with other elements in the commonSA namespace, like the Parameter element, which could omit the Value, CaptionID, DescID, and Access elements from the parameter itself. For the current DUT 1 configuration, this change alone could reduce 15072 rows. The viability of changing the schema is low, as it would trigger updates for all devices, each device's communication package, and any tooling that parses the common substation automation format.

These private tags could also be implemented next to the LD, LN, DO, and DA tags, rather than in a separate section within the IED. This would be beneficial, as the private data tags already mimic the IED data model. With this change, the size of the deduplicated data would shrink significantly.

References

ABB. (2024). *REX615 Functions*. https://library.e.abb.com/public/d415200b4dc44b86952d63b78e12ebaf/REX615_functions_2NGA001862_ENb.pdf?x-sign=RGNwHXrwsM1GD9OoQfNltU01PkCsL8fLqgUqKf8KYCqzCK4Wh0w5AFPCO24oiCid

ABB. (2025a). *Protection and control REX615*. Medium Voltage Products. https://new.abb.com/medium-voltage/digital-substations/protection-relays/multiapplication/protection-and-control-rex615

ABB. (2025b). *REX615 Connectivity Package 6.0.2* (Version 6.0.2) [English]. ABB. https://search.abb.com/library/Download.aspx?DocumentID=9AKK108470A3380&LanguageCode=en&DocumentPartId=&Action=Launch

Akinshin, A. (2019). *Pro .NET Benchmarking: The Art of Performance Measurement*. Apress.

Baumstark, A., & Pohl, C. (2019). Lock-free Data Structures for Data Stream Processing: A Closer Look. *Datenbank-Spektrum*, *19*(3), 209–218. https://doi.org/10.1007/s13222-019-00329-4

Busatto, G., Lohrey, M., & Maneth, S. (2005). Efficient Memory Representation of XML Documents. In G. Bierman & C. Koch (Eds), *Database Programming Languages* (pp. 199–216). Springer. https://doi.org/10.1007/11601524_13

Carter, P. (2020, April 29). Introducing C# Source Generators. *..NET Blog*. https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (Fourth Edition). The MIT Press.

Fredkin, E. (1960). Trie memory. *Commun. ACM*, *3*(9), 490–499. https://doi.org/10.1145/367390.367400

Ganss, M. (2026). *Mganss/XmlSchemaClassGenerator* [HTML]. https://github.com/mganss/XmlSchemaClassGenerator (Original work published 2013)

IEC. (n.d.-a). *IEC 61850 – Home*. Retrieved 10 January 2026, from https://iec61850.dvl.iec.ch/

IEC. (n.d.-b). *IEC 61850 technical principles*. Retrieved 24 February 2026, from https://iec61850.dvl.iec.ch/what-is-61850/technical-principles/

IEC. (n.d.-c). *Mapping platform | IEC 61850*. Retrieved 24 February 2026, from https://mapping.iec.ch/#/maps/21

IEC. (2013). *Introduction and overview* (Version Consolidated, 2.0, Pt 1) [Technical Report].

IEC. (2020a). *Basic information and communication structure – Abstract communication service interface (ACSI)* (Version Consolidated, 2.1, Pt 7-2) [International Standard].

IEC. (2020b). *Specific communication service mapping (SCSM) – Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3* (Version Consolidated, 2.1, Pt 8-1) [International Standard].

IEC. (2020c). *Specific communication service mapping (SCSM) – Sampled values over ISO/IEC 8802-3* (Version Consolidated, 2.1, Pt 9-2) [International Standard].

IEC. (2022). *Communication requirements for functions and device models* (Version Consolidated, 2.1, Pt 5) [International Standard].

IEC. (2024). *Configuration description language for communication in power utility automation systems related to IEDs* (Version Consolidated, 2.2, Pt 6) [International Standard].

Jang, B., Abubakari, A., & Kim, N. (2018). IEC 61850 SCL Validation Using UML Model in Modern Digital Substation. *Smart Grid and Renewable Energy*, *9*(8), 127–149. https://doi.org/10.4236/sgre.2018.98009

Kuszmaul, W., & Xi, Z. (2024). Towards an Analysis of Quadratic Probing. In K. Bringmann, M. Grohe, G. Puppis, & O. Svensson (Eds), *51st International Colloquium on Automata, Languages, and Programming (ICALP 2024)* (Vol. 297, p. 103:1-103:19). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ICALP.2024.103

Lukka, K. (2003). The Constructive Research Approach. In *Case Study Research in Logistics* (pp. 83–101).

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, *56*(5), 9:1-9:13. https://doi.org/10.1147/JRD.2012.2204512

Microsoft. (n.d.-a). *XmlDocument Class (System.Xml)*. Retrieved 12 April 2026, from https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmldocument?view=net-10.0

Microsoft. (n.d.-b). *XmlReader Class (System.Xml)*. Retrieved 1 March 2026, from https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmlreader?view=net-10.0

Microsoft. (n.d.-c). *XmlSerializer Class (System.Xml.Serialization)*. Retrieved 1 March 2026, from https://learn.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=net-10.0

Microsoft. (2021, September 15). *Hashtable and Dictionary Collection Types—.NET*. https://learn.microsoft.com/en-us/dotnet/standard/collections/hashtable-and-dictionary-collection-types

MZ Automation GmbH. (2026). *Mz-automation/libiec61850* [C]. https://github.com/mz-automation/libiec61850 (Original work published 2017)

.NET Platform. (2026). *Dotnet/runtime* [C#]. https://github.com/dotnet/runtime (Original work published 2019)

Newton-Evans Research. (2019, April 1). 94% of North American Electric Utilities Surveyed Use DNP3 for SCADA. *Newton-Evans Research Company, Inc.* https://www.newton-evans.com/94-of-north-american-electric-utilities-surveyed-use-dnp3-for-scada/

Nicola, M., & John, J. (2003). *XML parsing: A threat to database performance*. 175–178. https://doi.org/10.1145/956863.956898

Pan, Y. (2011). Research on IED Configurator Based on IEC 61850. *2011 International Conference on Control, Automation and Systems Engineering (CASE)*, 1–4. https://doi.org/10.1109/ICCASE.2011.5997664