LAPIN AMK
Lapland University of Applied Sciences

PARSING STANDARDIZED FORESTRY DATA IN RUST

Case Study of the ClimateForest Project

Bäckman, Marc

Thesis
Information and Communication Technology
Bachelor of Engineering

2025

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön tiivistelmä

Tieto- ja viestintätekniikan koulutus
Insinööri (AMK)

Tekijä	Marc Bäckman	Vuosi	2025
Ohjaaja	Tauno Tepsa		
Toimeksiantaja	Lapin ammattikorkeakoulu		
Työn nimi	Standardoitujen metsävaratietojen jäsentäminen Rust-ohjelmointikielellä: Tapaustutkimus ClimateForest-projektista		
Sivumäärä	41 + 4		

Tämä opinnäytetyö keskittyy standardoitujen metsävaratietojen jäsentämiseen ja hyödyntämiseen Rust-ohjelmointikielellä osana ClimateForest-projektia. Työn päätavoitteena oli kehittää XML-jäsennystyökalu metsävaratietojen käsittelyyn. Nämä tiedot sisältävät yksityiskohtaisia tietoja metsäkuvioista, puuston ominaisuuksista, maantieteellisistä tiedoista ja metsäoperaatioista. Työkalu luo tietorakenteita Rust-ohjelmointikielellä mahdollistaen niiden integroinnin sovelluksiin tiedon analysointia ja hallintaa varten.

Opinnäytetyössä kehitettiin kolme projektia: ensimmäisessä hyödynnettiin olemassa olevaa Rust-kirjastoa XML-datan muuntamiseen tietorakenteiksi, toisessa kehitettiin oma kirjasto samaan tarkoitukseen, ja kolmannessa pyrittiin muuntamaan XSD-tiedostot tietorakenteiksi. Toisessa projektissa onnistuttiin tavoitteiden mukaisesti, ja se julkaistiin Rust-kirjastona sekä dokumentoitiin.

Tämä opinnäytetyö tarjosi tilaisuuden syventää Rust-ohjelmointikielen osaamista sekä hankkia kokemusta kirjaston testaamisesta, julkaisemisesta ja dokumentoinnista.

Avainsanat jäsentäjä, Rust, XML

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Study Programme in Information
and Communication Technology
Bachelor of Engineering

Author	Marc Bäckman	Year	2025
Supervisor	Tauno Tepsa		
Commissioned by	Lapland University of Applied Sciences		
Title of Thesis	Parsing Standardized Forestry Data in Rust: A Case Study of the ClimateForest Project		
Number of pages	41 + 4		

This thesis focuses on parsing and utilizing standardized forestry data using the Rust programming language within the ClimateForest project. The primary goal was to develop an XML parsing tool to process forestry data, which includes detailed information on forest stands, tree characteristics, geographic data, and forestry operations. The tool will generate structured data representations in Rust, enabling integration with applications for data analysis and management.

To achieve this, three projects were developed: the first utilized an existing Rust crate to convert XML into data structures, the second involved developing a custom library crate to perform the same task, and the third attempted to convert XSD files into data structures. The second project successfully met its goals and was published as a Rust crate with proper documentation.

This thesis provided a valuable opportunity to deepen knowledge of the Rust programming language while gaining experience in testing, publishing, and documenting a Rust crate.

Key words parser, Rust, XML

CONTENTS

1 INTRODUCTION

Forests are crucial to environmental health and provide ecological, economic, and social benefits. However, the impacts of climate change, including diseases and wildlife damage, threaten their sustainability. The *ClimateForest* project addresses these challenges by developing tools for forest damage prevention. Work Package 4 (WP4) in the *ClimateForest* project focuses on creating 2D and 3D visualizations of climate change effects on forests, especially damages caused by **pine blister rust** and **moose browsing**. To achieve this, large amounts of forestry data must be processed efficiently for visualization applications. (Lapland University of Applied Sciences, n.d.)

The data is provided in XML format. It contains details about forest stands, tree species, soil conditions, geographic information (GML), and forestry operations. The data must be parsed from the XML format into Rust data structures, that can then be used in the application developed in WP4. This thesis examines the role of the Rust programming language in parsing the forestry data efficiently from XML. Rust offers memory safety, high performance, and concurrency features, making it an ideal choice for handling complex and large-scale datasets.

Methodology:

1. The forestry data is obtained from *Sitowise*, a company providing XML files and an API with structured forestry information.
2. An XML parsing tool is implemented in Rust, utilizing libraries such as *serde* and *quick-xml* for deserialization.
3. Extracted data is structured into **Rust data types** and converted into formats compatible with the visualization applications.
4. The tool is tested with various XML datasets to ensure accuracy, efficiency, and compatibility with WP4 visualization tools.

By developing tools and systems using Rust, this study contributes to the broader goal of improving forest management practices through efficient data utilization and visualization.

In this thesis, ChatGPT version 4 was used in the final stages for language editing.

2 CLIMATE FOREST PROJECT

2.1 ClimateForest

The objective of the *ClimateForest* project is to develop tools for preventing forest damage. It is financed by Interreg Aurora, the European Union, and the Regional Council of Lapland. The project is split into five work packages (WP).

- WP1: Data gathering and requirement definition on forest damaging agents
- WP2: Development of a climate change risk management and prediction model for pine blister and moose browsing damage
- WP3: A system for long-term forest damage and disease identification
- WP4: Develop a method to visualize comprehensive climate change adaptation data and effects
- WP5: Dissemination and knowledge sharing in the Aurora region

2.2 Work Package 4

Work Package 4 in the *ClimateForest* project focuses on visualizing climate change effects and damages to the forest. The main effects that will be visualized are **pine blister rust** and **moose browsing damages**. Pine blister rust is a fungal disease that infects pine trees, causing the formation of yellow or orange blisters. It damages the tree's branches and trunk.

The main objective of WP4 is to visualize forestry data in both 2D and 3D views, incorporating simulations of climate change effects. For instance, trees infected with pine blister rust will be displayed in a different color from the healthy trees in the 2D view. In the 3D view, the models of infected trees will differ from those of healthy trees.

Web-XR implementations will be developed to support the 3D visualizations, allowing users to select an area from the 2D map and project it into 3D. Additionally, users will be able to visualize the effects of forest operations, such as cutting or thinning, in both 2D and 3D views.

To generate these visualizations, forestry data is either read from XML files or retrieved from an API that provides the data in XML format.

2.3 The Data

The data used in WP4 is provided by Sitowise, a Finnish smart city company offering services in real estate, infrastructure, and digital solutions in both Finland and Sweden. Sitowise also provides forestry-related services, including XML files and an API containing forestry data. The XML data typically consists of real estate properties, which are divided into parcels, and further subdivided into forest stands. The stand data includes detailed information about the forest, which is essential for the application developed in WP4.

Stand information consists of the different species of trees and their respective amounts of trees in the stand. Also, characteristics of the trees, such as mean diameter, mean age, basal area, and mean height are included. Other information about the stand includes forestry operations done on the stand, such as cutting or thinning along with their planned dates. There is also a basic data section that tells us about the stand’s area, fertility class, soil type, drainage state, main tree species, etc.

```
<st:Stand id="2143884" realEstateId="25476" parcelId="0">
  <st:StandBasicData>
    <st:StandNumber>1109</st:StandNumber>
    <st:MainGroup>1</st:MainGroup>
    <st:SubGroup>1</st:SubGroup>
    <st:FertilityClass>4</st:FertilityClass>
    <st:SoilType>21</st:SoilType>
    <st:DrainageState>1</st:DrainageState>
    <st:DevelopmentClass>T1</st:DevelopmentClass>
    <st:StandQuality>10</st:StandQuality>
    <st:MainTreeSpecies>1</st:MainTreeSpecies>
    <st:Accessibility>2</st:Accessibility>
    <st:StandBasicDataDate>2024-11-20</st:StandBasicDataDate>
    <st:StandInfo/>
    <st:Area>5.214</st:Area>
    <gdt:PolygonGeometry>
      <gml:polygonProperty xlink:type="simple">
        <gml:Polygon srsName="EPSG:3067">
          <gml:exterior>
            <gml:LinearRing>
            ...
            </gml:LinearRing>
          </gml:exterior>
        </gml:Polygon>
      </gml:polygonProperty>
    </gdt:PolygonGeometry>
  </st:StandBasicData>
  <ts:TreeStandData>
    <ts:TreeStandDataDate type="1" date="2024-11-20">
      <tst:TreeStrata>
          <tst:TreeStratum id="360684736">
          <tst:StratumNumber>0</tst:StratumNumber>
          <tst:TreeSpecies>1</tst:TreeSpecies>
          <tst:Storey>1</tst:Storey>
          <tst:Age>5</tst:Age>
          <tst:StemCount>2100</tst:StemCount>
          <tst:MeanHeight>0.5</tst:MeanHeight>
          </tst:TreeStratum>
      </tst:TreeStrata>
    </ts:TreeStandDataDate>
  </ts:TreeStandData>
</st:Stand>
```

Listing 1. Example of Stand data

The stand's geographic information is noted by GML (Geographic Markup Language). The structure of the GML can vary, depending on the XML file's schema version, but its purpose is always the same: It displays the coordinates of the polygon enclosing the area of the stand. In the Finnish forestry data XML files that come from Sitowise, the coordinates are in **EPSG:3067** format and the units of the coordinates are in meters.

2.3.1 GML

GML, or Geography Markup Language is "the XML grammar defined by the Open Geospatial Consortium (OGC) to express geographical features." (Wikipedia 1, n.d., para. 1). The purpose of GML in this case is to display the geometry of the stand as a polygon enclosing the area. The polygon consists of an external *LinearRing* of coordinates that represent the outer boundaries of the stand and optional interior *LinearRings* that represent enclosed areas within the stand that are excluded from it.

```
<gdt:PolygonGeometry>
  <gml:polygonProperty xlink:type="simple">
    <gml:Polygon srsName="EPSG:3067">
      <gml:exterior>
        <gml:LinearRing>
            <gml:posList srsDimension="2">...</gml:posList>
        </gml:LinearRing>
      </gml:exterior>
    </gml:Polygon>
  </gml:polygonProperty>
</gdt:PolygonGeometry>
```

Listing 2. Example of GML

In the above example, the element **<gml:posList srsDimension="2">** contains the coordinates of a polygon in two dimensions. The outer boundary of a forest stand is noted by the **<gml:exterior>** tags.

2.3.2 EPSG Geodetic Parameter Dataset

An EPSG (European Petroleum Survey Group) registry "is a public registry of geodetic datums, spatial reference systems, Earth ellipsoids, coordinate transformations and related units of measurement." (Wikipedia 2, n.d., para. 1). It was created in 1985 to standardize and share spatial data between members of the European Petroleum Survey Group. EPSG codes vary between 1024 and 32767. The EPSG code for Finland is 3067.

EPSG coordinates can be expressed in either degrees (angular coordinates) or in meters (linear coordinate). Finland uses linear coordinates and they are expressed in meters.

Many GIS (Geographic Information Systems), including the one being developed in ClimateForest WP4, use EPSG codes. The Finnish linear coordinates are retrieved from the XML file and projected into EPSG:4326 angular coordinates within the ClimateForest application. This projection ensures the data can be accurately displayed on a map. Without this projection, the polygon defined by the coordinates would become distorted, as representing a curved surface on a flat plane causes image stretching.

The coordinate system that displays the projected coordinates in EPSG:4326 onto a map is also referred to as WGS 84.

2.3.3 WGS 84

As explained in Wikipedia 3 (n.d.), WGS 84 (World Geodetic System 1984) is a standard used in cartography, geodesy, and satellite navigation, including GPS. The coordinate origin of WGS 84 is at the Earth's center of mass. WGS coordinates are expressed in longitude and latitude, using angles rather than meters. Latitude measures the angle north or south of the Equator, ranging from -90° (South Pole) to 90° (North Pole). Longitude measures the angle east or west of the Prime Meridian, ranging from -180° to 180°.

3 CONVERTING XML TO STRUCTS

3.1 XML

Before discussing how structured data representations are created from the XML data, it is important to explain what XML is and how it works. XML (Extensible Markup Language) is a markup language similar to HTML, but unlike HTML, it does not have predefined tags. While HTML uses tags like **<html>**, **<head>**, and **<body>**, XML allows users to define their own tags.

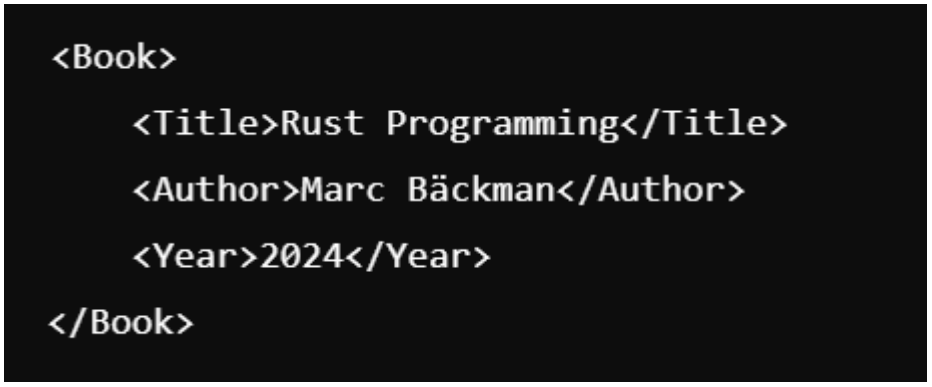

Figure 1. Example of XML representing a book

Here the user has defined the tags **<Book>**, **<Title>**, **<Author>**, and **<Year>**.

3.2 XSD

XSD stands for XML Schema Definition. It is a language used to define the structure, content, and data types of an XML document, serving as a blueprint for XML files. All XML documents containing the same type of data must follow the rules defined in their corresponding XSD.

The previous XML example would have the following XSD definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <!-- Root element definition -->
    <xs:element name="Book">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Title" type="xs:string" />
                <xs:element name="Author" type="xs:string" />
                <xs:element name="Year" type="xs:gYear" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 2. Example of XSD representing a book.

In the above XSD is a complex type “Book” that contains three elements: “Title”, “Author”, and “Year”. The element’s types are also mentioned. “Title” and “Author” are string types and “Year” is of type *gYear*, meaning that it is a date type, but it only contains the year of the date.

After an XSD file specifying the rules for the XML document is created, it is referenced to in the XML using the **xsi:schemaLocation** attribute in the root element. Here is the XML document referencing the XSD:

```xml
<Book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="book.xsd">
    <Title>Rust Programming</Title>
    <Author>Marc Bäckman</Author>
    <Year>2024</Year>
</Book>
```

Figure 3. XML for the book that is built from the XSD specification

3.3 Rust Struct

The goal is to generate structured data representations, or structs, in the Rust programming language from both XML and XSD files. Structs group data together

into a single data structure, combining fields of different types to create a custom type. The corresponding Rust struct for our example would look like this:

```
pub struct Book {
    pub title: String,
    pub author: String,
    pub year: u32,
}
```

Listing 3. Rust Struct representing the book

The structs can then be integrated into the application that visualizes and manages the data in WP4.

4 RUST PROGRAMMING LANGUAGE

“Rust is a general-purpose programming language emphasizing performance, type safety, and concurrency. It enforces memory safety, meaning that all references point to valid memory. It does so without a traditional garbage collector; instead, memory safety errors and data races are prevented by the "borrow checker", which tracks the object lifetime of references at compile time.” (Wikipedia 4, n.d., para. 1). This description includes the main reasons why Rust is a suitable programming language for parsing large XML files that contain large amounts of data. The key points will be explained in more detail in the following sections.

There are many unique features in Rust that do not exist in other programming languages. Some of the features that make Rust stand out from the rest of the programming languages include its ownership model, the borrow checker, and guaranteed memory safety without garbage collection.

This section will discuss the key Rust features that are essential for developing an XML parser.

4.1 Ownership and borrowing

Rust uses an ownership and borrowing system to manage memory without a garbage collector. This means variables automatically clean up their memory when they are no longer needed. In an XML parser, this is particularly useful for handling large XML files efficiently.

Rust’s ownership model ensures memory safety by making sure that each piece of data has only one “owner” at a time. For example, when you assign a string to a variable, the variable owns the string. When the variable is no longer used, Rust frees the memory for that data.

In Rust, instead of transferring ownership, you can “borrow” the data by making a reference to it. A reference to a variable is noted by &. Here is an example of borrowing:

```rust
fn main() {
    let s: String = String::from("Hello, world!"); // `s` owns the String
    print_message(&s); // Borrow `s` by passing a reference

    // We can still use `s` because it was borrowed, not moved
    println!("Original message: {}", s);
}

fn print_message(message: &String) {
    println!("Message: {}", message); // Using the borrowed reference
}
```

Figure 4. Example of borrowing a string

Normally in Rust, the ownership of the variable ‘s’ would be transferred to the *print_message* function, meaning it couldn’t be accessed again. However, by borrowing the variable, we can "look" at its data within the *print_message* function without taking ownership. As a result, we can still access ‘s’ later in the code. (Rust Team, n.d. Chapter 4)

This means that you can pass large amounts of data into a function without duplicating its memory. For example, using a reference **&str** lets you manipulate or parse the XML content directly from the source, without the need to copy it into a new variable.

Rust’s references are similar to pointers in C / C++, but they have some main differences:

- Rust references are guaranteed to be safe at compile time because Rust does not allow null references. Options are used instead.
- Rust uses automatic memory management. This ensures that dangling references (references to free memory) cannot exist.

4.2 Pattern Matching

XML consists of start tags, empty tags, text, and end tags. To handle each of these cases, we can use Rust’s pattern matching feature. Pattern matching is like the switch-case in other programming languages, but it is a lot more expressive.

Rust's match allows us to access the data, even when it is encapsulated within a complex type like Option or Result. (Rust Team, n.d. Chapter 6)

4.3 Options

In Rust, **Option** is used to represent values that may or may not be present. It is commonly used in situations where the existence of data is uncertain. Option helps avoid null pointer errors which are common in other languages. An Option is noted by **Option<T>** and **None**.

Optional data can be manipulated by first inspecting if the option contains data or not with a match statement.

Example with pattern matching and Option:

```
// Option containing a value
let value: Option<i32> = Some(42);

match value {
    Some(number: i32) => {
        println!("The value is: {}", number); // Access the data inside `Some`
    }
    None => {
        println!("No value found!"); // Handle the case where it's `None`
    }
}
```

Figure 5. Example of using match statement with optional data

In the first two projects in this thesis, all the fields of the generated structs are defined as optional, since the XML file may or may not contain data for a specific tag.

4.4 Structs

In Rust, a struct is a data type that allows you to group different types of values into a single type. Each value inside a struct is called a field, and each field can have a different type. Structs are like classes in object-oriented languages, but they don't contain methods. Instead, methods for structs are defined within an **impl** block. (Rust Team, n.d. Chapter 5)

Example of struct:

```
pub struct Rectangle {
    width: u32,
    height: u32,
}

// Implementation block
impl Rectangle {

    // A method to calculate the area of the rectangle
    pub fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Listing 4. A Struct with an implementation

The purpose of the XML parser is to generate data types, or structs, from the XML file, allowing the data to be used within a Rust program. Once the structs are generated, developers can work with the data by accessing fields, performing operations, and serializing or deserializing it when necessary. The parser itself will not implement methods for the structs; this task will be left to developers during the development of the WP4 application.

4.5 Rust Compiler

Rust's compiler is one of the language's strongest features. It will not let your program compile if it contains errors. It catches mistakes at compile time, which reduces the likelihood of runtime bugs and crashes. This forces developers to write cleaner code.

One of the most notable features of the compiler is its error messages. These messages are incredibly helpful, as they not only highlight the problematic lines of code but also suggest ways to fix the issues. As a result, reading error messages often leads to learning something new about the language. The compiler marks erroneous code in red and provides a diagnostic message for it. The diagnostic message can be accessed directly by clicking the code.

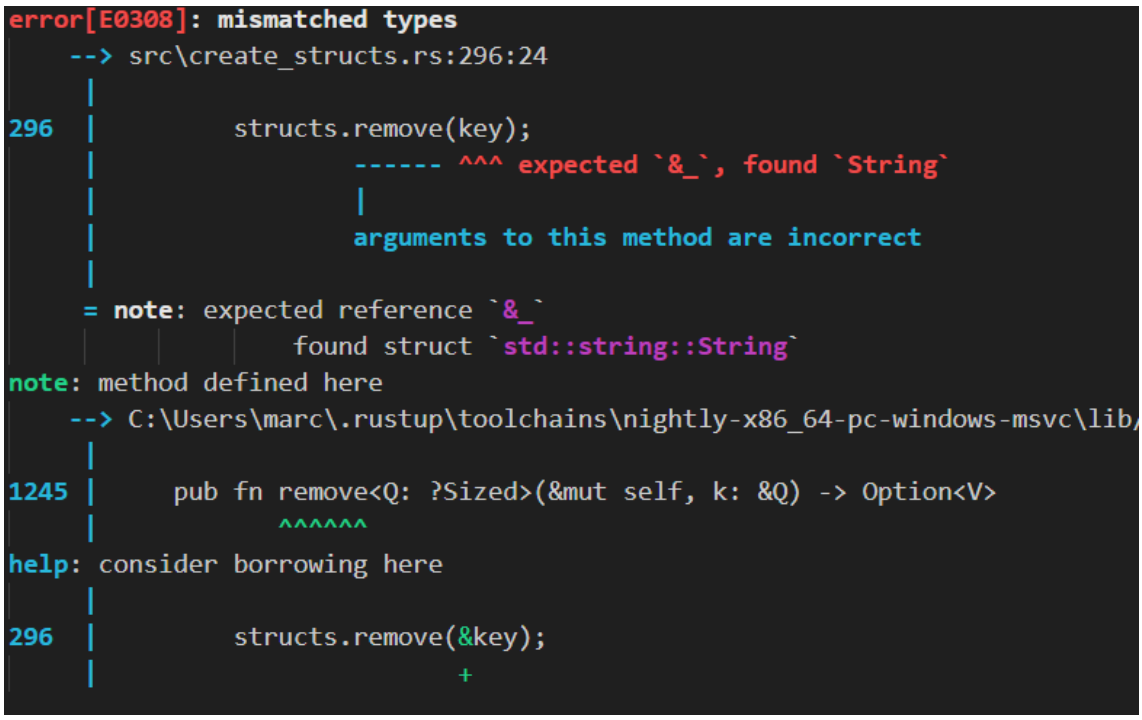

Figure 6. Example of Rust compiler's diagnostic message

In this example, the compiler tells the user that there are mismatched types. The code has a variable of type String, when the variable should be a reference to a String. The compiler then suggests how to fix the problem by borrowing the variable with a reference. If Rust's ownership and borrowing rules are ignored, the code will not be compiled.

For developers, the Rust compiler is more than just a tool for flagging errors. It serves as an interactive assistant for learning and debugging.

4.6 Crates

Crates in Rust are like modules or libraries in other programming languages. Crates can be compiled and shared. A library crate for a parser that produces reusable code for the developers in WP4 to use, will be published. Publishing a crate is discussed in more detail in chapter 7.

5 PROJECT DESCRIPTIONS

Three separate projects were developed for this thesis, all aiming to do the same thing: generating structs for manipulating XML forestry data.

1) A Rust crate (library) called *xml_schema_generator* was used to automatically generate Rust structs from an XML file.
2) A custom XML schema generator with the same objective as *xml_schema_generator* was developed.
3) Generating structs directly from XSD files was attempted.

5.1 General Workflow

Rust has a steep learning curve due to its unique features that are not commonly found in other languages. Since Rust was unfamiliar at the start of this project, guidance from ChatGPT was used during code development. However, this does not imply that ChatGPT generated all the code—far from it. The free version of ChatGPT, based on model 3.5, was used, with occasional access to model 4.0 for limited queries.

Most of the time, ChatGPT's generated code did not produce the desired outcome and had to be modified multiple times to work correctly. As a general workflow, ChatGPT provided a starting point—a code snippet that could be refined and built upon until it functioned properly. Additionally, ChatGPT was particularly helpful when dealing with complex syntax, such as closures and filtering or mapping operations. GitHub Copilot also played a role by assisting with syntax and autofilling code.

The code is heavily commented to keep track of functionality and logic throughout development. GitHub Copilot's suggested comments were particularly useful, and I kept many of them as guidance.

As mentioned earlier, the Rust compiler and its error messages were also valuable tools in the development process. Together with ChatGPT and GitHub

Copilot, they made learning and programming in Rust significantly easier, even for a beginner like me.

5.2 Implementation

5.2.1 A Project that uses the *xml_schema_generator* crate

The first project uses a Rust library, or crate, called *xml_schema_generator*. It generates structs for the data directly from an XML through a process called deserialization. A separate project was created for generating structs from an XML file and from the XML fetched from the API. Because the file-based data contains the element *RealEstates* and the API data contains the element *Stands,* both under their common parent element *ForestPropertyData*, the corresponding Rust struct includes optional fields for *RealEstates* and *Stands*.

```rust
#[derive(Serialize, Deserialize)]
3 implementations
pub struct ForestPropertyData {
    #[serde(rename = "@xmlns")]
    pub xmlns: String,
    #[serde(rename = "@xmlns:re", skip_serializing_if = "Option::is_none")]
    pub xmlns_re: Option<String>,
    #[serde(rename = "@xmlns:st")]
    pub xmlns_st: String,
    #[serde(rename = "@xmlns:ts")]
    pub xmlns_ts: String,
    #[serde(rename = "@xmlns:tst")]
    pub xmlns_tst: String,
    #[serde(rename = "@xmlns:dts")]
    pub xmlns_dts: String,
    #[serde(rename = "@xmlns:tss")]
    pub xmlns_tss: String,
    #[serde(rename = "@xmlns:op")]
    pub xmlns_op: String,
    #[serde(rename = "@xmlns:sf")]
    pub xmlns_sf: String,
    #[serde(rename = "@xmlns:gdt")]
    pub xmlns_gdt: String,
    #[serde(rename = "@xmlns:co")]
    pub xmlns_co: String,
    #[serde(rename = "@xmlns:gml")]
    pub xmlns_gml: String,
    #[serde(rename = "@xmlns:xsi")]
    pub xmlns_xsi: String,
    #[serde(rename = "@xmlns:xlink")]
    pub xmlns_xlink: String,
    #[serde(rename = "@schemaLocation")]
    pub xsi_schema_location: String,
    #[serde(rename = "@schemaPackageVersion", skip_serializing_if = "Option::is_none")]
    pub schema_package_version: Option<String>,
    #[serde(rename = "@schemaPackageSubversion", skip_serializing_if = "Option::is_none")]
    pub schema_package_subversion: Option<String>,
    #[serde(rename = "$text")]
    pub text: Option<String>,
    #[serde(rename = "RealEstates", skip_serializing_if = "Option::is_none")]
    pub re_real_estates: Option<ReRealEstates>,
    #[serde(rename = "Stands", skip_serializing_if = "Option::is_none")]
    pub st_stands: Option<StStands>
}
```

Figure 7. *ForestPropertyData* struct

It is important to note that the last two fields must be optional because the data structure differs between the API and the XML file. *ForestPropertyData* can contain either *RealEstates* or *Stands,* but not necessarily both.

In the schema documentation, its structure looks like this:

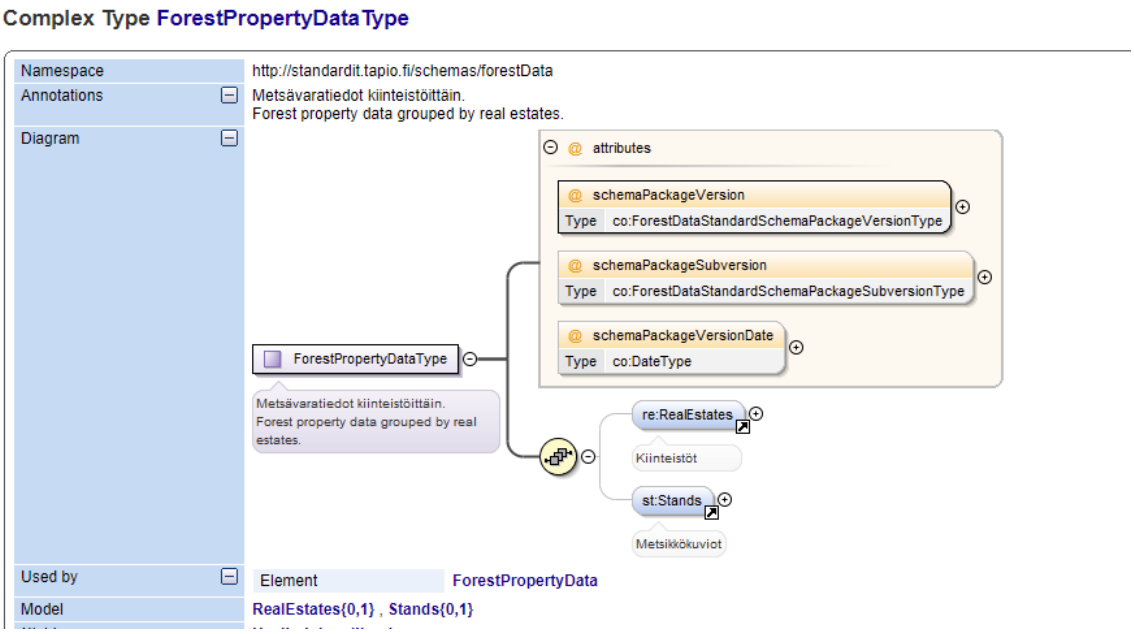

Figure 8. Sitowise Documentation for *ForestPropertyData* schema

Structs were generated into a single file, combining both the file-based and API-based structs. Fields with the same name were merged by comparing their definitions. If a field was present in one source but missing in the other, it was marked as optional. Once all the structs were merged into a single file, they were ready for use, allowing the same data structures to work with XML from both a file and an API.

A project called *forestry_xml_parser* contains this file with the structs. This project was created specifically for testing the structs. The repository for the project is available at GitHub. The structs were tested with multiple XML files and by reading XML from the API. Additionally, *serde* was used to convert XML data into JSON files using these structs.

Serde is a framework in Rust for serializing and deserializing data. Serialization converts a Rust data structure into a storable or transmittable format, such as JSON or XML. Deserialization is the reverse process—transforming data from formats like JSON or XML back into Rust data structures.

The JSON files were successfully generated from the XML files. The next step was to verify the conversion by converting the JSON back to XML and comparing it to the original XML. Conversion from JSON to XML was done using two crates: *quick-xml* for generating the XML and *xmlem* for pretty-printing the XML (adding indentations). However, the resulting XML data between tags was printed on separate lines, which made it differ from the original XML.

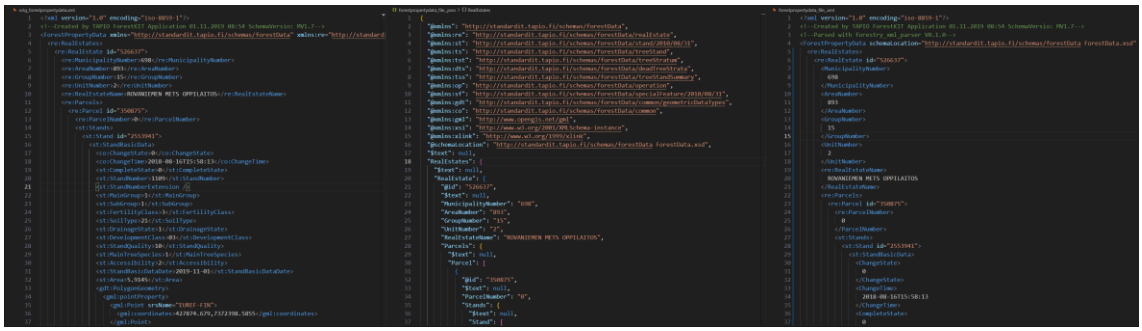

Figure 9. Workflow from original XML to JSON and back to XML

The workflow illustrates the process: the original XML is on the left, the converted JSON is in the middle, and the XML resulting from the conversion back from JSON is on the right. Note that the XML on the right differs from the original due to how *xmlem* adds new lines after each tag. The namespace prefixes in the resulting XML were manually inserted by a function, as *xmlem* does not handle namespaces. Additionally, information about the parser version used for the conversion was included in the resulting XML: **<!--Parsed with forestry_xml_parser V0.1.0-->**.

Next, a custom “xml_schema_generator” was developed, which would dynamically generate data structures by reading an XML file.

5.2.2 Custom Schema Generator

A project called *schema_generator* was created to replicate the functionality of *xml_schema_generator*. In other words, it should generate data structures dynamically by reading an XML file. The goal was to explore the process of building such a generator.

In this project, a custom function to generate XML data from JSON was also developed. The repository for this project can be found at GitHub

To generate data structures from XML, it is important to understand how XML data is structured. XML consists of start tags, empty tags, end tags, and content. Below are the basic components of XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<parent> <!-- Parent tag -->
    <child> <!-- Start tag -->
        Child content here
    </child> <!-- End tag -->
    <empty_child /> <!-- Empty tag -->
</parent>
```

Figure 10. Basic XML structure

The names inside the tags are called elements. This XML has the elements “parent”, “child” and “empty_child”. The element “child” contains a String content.

The corresponding struct for the above XML would look like this:

```
#[derive(Debug, Serialize, Deserialize)]
struct Parent {
    child: Option<String>,        // The content inside <child>
    empty_child: Option<String>,  // Representing the <empty_child /> tag
}
```

Figure 11. Corresponding Struct for the XML in Figure 13

In the generated structs, all fields are considered optional. For instance, not every field named "child" will necessarily have a String value. This is how *schema_generator* operates as well—it assumes that every field is optional. By treating all fields as optional, it helps prevent bugs in cases where the XML file may not always contain the same child tags under a specific parent tag.

To generate structs from XML, a reader processes the XML file element by element. The reader identifies the start tags, empty tags, and end tags, then creates the structs based on the parent-child relationships between the elements.

The first step is to create the structs that will generate the resulting data structures:

```
#[derive(Debug, Clone)]
pub struct XMLField {
    pub name: String,
    pub field_type: String,
}

#[derive(Debug)]
pub struct XMLStruct {
    pub name: String,
    pub fields: Vec<XMLField>,
}

impl Clone for XMLStruct {
    fn clone(&self) -> Self {
        XMLStruct {
            name: self.name.clone(),
            fields: self.fields.clone(),
        }
    }
}
```

Listing 5. The Structs used in *schema_generator*

The Algorithm for Generating Structs from XML

Imagine reading an XML file element by element. For the first element, you create a struct. You will notice that every nested element inside the parent element becomes a field in the parent struct. Elements further nested within those will become fields for their respective structs, and so on. The final struct is only created when the end tag for an element is encountered. This process is managed by tracking two HashMaps: one for a stack of structs and one for the finalized structs. The stack of structs is built from each start tag, and the finalized struct is created when the corresponding end tag is encountered.

The algorithm in its simplest form looks like this:

```rust
fn create_structs(reader: &mut Reader<&[u8]>) -> HashMap<String, XMLStruct> {

    // Stack of structs being constructed
    let mut stack: Vec<XMLStruct> = Vec::new();

    // Finalized structs
    let mut structs: HashMap<String, XMLStruct> = HashMap::new();

    loop {
        match reader.read_event() {

            // Handle a start tag:
            Ok(Start(ref e)) => {
                // Create a new struct for this element
                // Push the struct onto the stack
                // If there's a parent struct, add this struct as a field to the parent struct
            },

            // Handle a self-closing tag:
            Ok(Empty(ref e)) => {
                // Create a new struct for this element
                // If there's a parent struct, add this struct as a field to the parent struct
            },

            // Handle an end tag:
            Ok(End(ref e)) => {
                // Pop the current struct from the stack
                // Update the finalized structs with the current struct
                // Merge fields if the struct already exists in the finalized collection of structs
            },

            // End of XML, exit the loop
            Ok(Eof) => break,

            // Handle parsing errors
            Err(e) => panic!("Error at position {}: {:?}", reader.buffer_position(), e),

            // Handle other events
            _ => (),
        }
    }

    structs
}
```

Figure 12. Algorithm to generate structs from XML tags

There are additional things to consider than just creating the parent structs, adding them to a stack, popping them from a stack, and adding the child structs as fields to the parent struct. However, those details are will not be commented in this section. Some of these things include keeping track of the count of fields and the maximum count of fields per struct, a HashMap of self-closing tags, and a Vector of start tags. The algorithm also involves functions for parsing attributes, removing structs without fields, adding empty structs, and updating field types. For more details on the algorithm, you can look at my code at GitHub

Writing the Structs to a File

Once all the finalized structs have been generated, a string representing those structs is created and written to a file. To generate this string, the contents of the structs are displayed in their string form. The goal is to produce a string that is compatible with the Serde library for serialization and deserialization. Commented code for generating the string is available at GitHub

When generating the string, all the attribute strings are generated as non-optional, except for the attribute *srsName*, which appears in the *GmlPolygon* and *GmlMultiPolygon* structs. All regular fields within the struct and text fields are optional. The text fields are optional as most elements do not contain any text content.

Once the string is created, it is saved into a file that defines the data model for the XML structure. With the structs defined, it becomes possible to read XML files and generate JSON files from the XML data.

After generating the JSON, it should be converted back into XML for comparison with the original XML. Unlike in the previous project, *quick-xml* and *xmlem* were not used for this conversion. Instead, a custom function was developed to convert JSON back into XML. This custom function ensures that the namespace prefixes are preserved from the original data.

Converting JSON back to XML

The custom function for converting JSON back to XML does not rely on the structs that are generated and saved to a file, nor does it use Serde in any form. Instead, it simply reads the JSON and converts it back to XML using recursion.

Recursion means that the function calls itself during its execution. Recursive functions are often used when handling nested structures, such as folders, subfolders, and files. In this case, the nested structure is the JSON structure of the forestry data.

The JSON structure contains objects that may themselves contain other objects or arrays. The recursive function processes the current object and then calls itself to handle any nested objects or arrays within that object.

One downside of using a recursive function is the potential loss of element order. The function processes sibling elements only after all nested elements of the current element have been handled. As a result, the new XML data may not retain the same order as the original.

Testing

Unit tests were written for some of the simpler functions that handle strings, while integration testing was used for the larger and more complex functions. Integration testing ensures that different parts of the system—such as XML parsing, struct generation, and JSON-to-XML conversion—work together correctly. For integration testing, various XML files were downloaded from *ForestKIT*, a Finnish forestry information system. *ForestKit* provides a user-friendly interface with a map, allowing users to select any number of forest stands and export them into an XML file.

Known Bugs

A bug was discovered during the testing of the JSON-to-XML conversion. If the first nested element has attributes and the parent element also has attributes, the nested element sometimes inherits the same prefix from the parent, instead of updating to its own prefix as intended. This issue is likely caused by the recursive nature of the function.

Another observation was that the resulting XML file had a different number of rows compared to the original one. This was not necessarily a bug, as the generated XML still contained all the relevant information from the original file. The difference in row count was due to certain *TreeStandDataDate* tags not containing any data within the tags. Additionally, in the original XML, GIS data

was written within multiple tags on a single row, whereas the recursive function generated a new row for each individual tag.

Limitations of *schema_generator*

The *schema_generator* generates structs directly from the XML file containing the data. This means that the generated structs are tailored specifically for that file. Consequently, using these structs to read other XML files—even those with similar data—may not work, as the new files could contain elements that were not present in the initial file from which the structs were generated.

While *schema_generator* works for a specific file, it is quite resource-intensive because it requires regenerating the structs for each new file. This constant regeneration consumes additional processing power and memory, and significantly increases development time. To address these issues, it would be more efficient to generate the structs directly from the XSD files instead of the XML files.

5.2.3 Schema Parser for XSD Files

As mentioned earlier, XSD files define the structure of XML files. The forestry data in the XML files for the *ForestPropertyData* struct is defined in the schema standard *Metsätietostandardiskeemat V33.03*. This schema standard consists of 184 XSD files and it follows the *StanForD 2010* forestry standard.

StanForD 2010

StanForD 2010 is the updated version of the *StanForD* standard that has been available since the end of the 1980s. It is a global forestry standard widely adopted by major manufacturers of forest machines, particularly those applying the cut-to-length (CTL) method.

The new version brings some new features to the existing standard. “StanForD 2010 introduces a concept for giving identities for machines, objects, stems, logs,

etc. with Keys and UserIds.” (StanForD 2010) Additionally, *StanForD 2010* now includes support for logging operations, production reporting, and operational monitoring. However, the most important feature of the 2010 update is its focus on reporting geographical information, or GIS data, which provides more detailed spatial context for forestry operations.

XSD Schema

Below is an example of a typical XSD file from the *Metsätietostandardiskeemat V33.03*.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://standardit.tapio.fi/schemas/forestData"
    xmlns="http://standardit.tapio.fi/schemas/forestData"
    xmlns:re="http://standardit.tapio.fi/schemas/forestData/realEstate"
    xmlns:st="http://standardit.tapio.fi/schemas/forestData/Stand"
    xmlns:co="http://standardit.tapio.fi/schemas/forestData/common" elementFormDefault="qualified"
    attributeFormDefault="unqualified" version="V20.01">

    <!-- Imports / Includes -->
    <xs:import namespace="http://standardit.tapio.fi/schemas/forestData/realEstate"
        schemaLocation="RealEstate.xsd"/>
    <xs:import namespace="http://standardit.tapio.fi/schemas/forestData/Stand"
        schemaLocation="Stand.xsd"/>
    <xs:import namespace="http://standardit.tapio.fi/schemas/forestData/common"
        schemaLocation="Common.xsd"/>

    <!-- Elementtimääritykset -->
    <!-- Element definitions -->
    <xs:element name="ForestPropertyData" type="ForestPropertyDataType"/>

    <!-- Tyyppimääritykset -->
    <!-- Type definitions -->
    <xs:complexType name="ForestPropertyDataType">
        <xs:annotation>
            <xs:documentation xml:lang="fi">Metsävaratiedot kiinteistöittäin.</xs:documentation>
            <xs:documentation xml:lang="sv"/>
            <xs:documentation xml:lang="en">Forest property data grouped by real estates.</xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element ref="re:RealEstates" minOccurs="0" maxOccurs="1">
                <xs:annotation>
                    <xs:documentation xml:lang="fi">Kiinteistöt</xs:documentation>
                    <xs:documentation xml:lang="sv"/>
                    <xs:documentation xml:lang="en"/>
                </xs:annotation>
            </xs:element>
            <xs:element ref="st:Stands" minOccurs="0" maxOccurs="1">
                <xs:annotation>
                    <xs:documentation xml:lang="fi">Metsikkökuviot</xs:documentation>
                    <xs:documentation xml:lang="sv"/>
                    <xs:documentation xml:lang="en"/>
                </xs:annotation>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="schemaPackageVersion"
            type="co:ForestDataStandardSchemaPackageVersionType" use="required"/>
        <xs:attribute name="schemaPackageSubversion"
            type="co:ForestDataStandardSchemaPackageSubversionType" use="optional"/>
        <xs:attribute name="schemaPackageVersionDate" type="co:DateType" use="optional"/>
    </xs:complexType>

</xs:schema>
```

Figure 13. ForestData.xsd

The XSD file consists of imports, element definitions, and type definitions. Imports are used to include referenced elements from other files in the schema, enabling modularity and reuse. Element definitions define new elements within the schema, specifying their structure and data types. Type definitions, on the other hand, define complex types, which are composed of multiple elements or references to other elements.

File Dependencies

XSD files often depend on other XSD files, creating a chain of dependencies. For the struct generator to function properly, it must know the field types of a struct before it can be created. This requires that all element types within an XSD file be resolved in advance. As a result, any imported XSD files must be processed first, before the main XSD file can be processed.

To organize the files in such a way that the dependencies are processed before the file itself, a topological sorting algorithm is applied to the files. Topological sorting is an algorithm that orders the vertices of a directed acyclic graph (DAG) such that for every directed edge u → v, vertex u comes before v in the ordering.

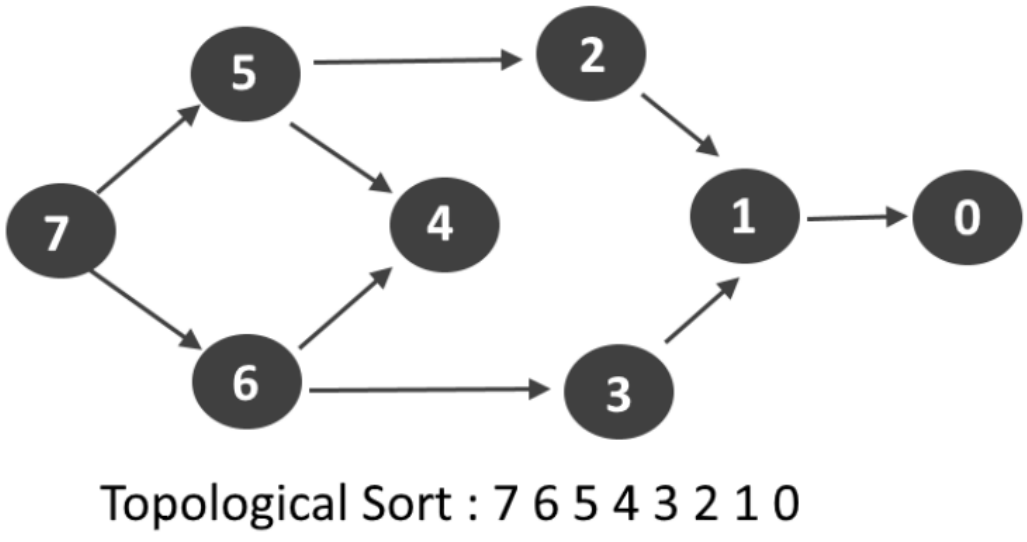

Figure 14. Example of topological sorting (OpenGenus, n.d.)

Note that the topological sort is not unique. For example, other possible ordering for the above example could be *{7, 5, 6, 4, 2, 3, 1, 0}* or *{7, 5, 6, 3, 2, 1, 0, 4}*.

In the context of XSD files:

- Each file is treated as a vertex.
- A directed edge *u* -> *v* exists if *v* depends on file *u*.

In Rust, topological sorting is implemented using a **HashMap** to track file dependencies, a **Vector** to store the sorted files, and two **HashSet** collections: one for visited files and one for temporary files. The file dependency HashMap keys represent the file name, and its values represent the corresponding imports to that file. Each file and its dependencies are processed in the sorting algorithm and the resulting Vector of files will be sorted in the correct order. For a more detailed explanation of the algorithm, refer to the comments in the GitHub repository.

The topological sorting algorithm does not handle cyclical graphs. In our case, three files had cyclic dependencies with others. To resolve this issue, the conflicting dependencies were removed, as they were deemed irrelevant for generating the *ForestPropertyData* data structure.

The Algorithm for Generating Structs from XSD

The next step proved to be quite challenging. With the files now sorted, the goal was to generate structs based on the XSD definitions. This process was approached similarly to generating structs from an XML file—by reading the file tag by tag. As the reader encounters a closing tag, it adds the child elements as fields to their respective parent element.

Conflicting Element Definitions

The main problem with generating structs from XSD files is managing conflicting element definitions across multiple files.

For example:

- File A might define an element **<ElementA>** as a string (xs:string)
- File B might define an element **<ElementA>** as a more complex type, involving more subtypes.

Because the files have different definitions for the same element, the element becomes ambiguous if its file of origin and context of use are not considered.

Ambiguities in Schema Imports and Prefixes

Another ambiguity that was discovered comes from the way schemas import element definitions from external files and use namespace prefixes locally.

For example:

- In **File A**, the prefix **co:** might refer to the external schema in **schema1.xsd**
- In **File B**, the prefix **co:** might refer to a completely different schema in **schema2.xsd**

The prefixes used in the imports are local to the schema file. Therefore, an element can be referred to with multiple different prefixes in different files.

For example:

- In File A, **<ElementA>** might be called **co:**ElementA
- In File B, **<ElementA>** might be called **wtc:**ElementA

To resolve these issues, it would be necessary to use FQNs (Fully Qualified Name) to keep track of the correct element definitions. A Fully Qualified Name is a unique name that is "complete in the sense that it includes (a) all names in the hierarchic sequence above the given element and (b) the name of the given element itself." (Weik, 2000)

For example:

- The element **co:ElementA** in **File A**, where **co:** refers to the namespace **http://example.com/schema1**, would be represented as ***{http://example.com/schema1}ElementA***.
- The element **co:ElementA** in **File B**, where **co:** refers to the namespace **http://example.com/schema2**, would be represented as ***{http://example.com/schema2}ElementA***.

A global FQN registry would keep track of these names, and each schema file would have their own HashMap that keeps track of the prefixes and their corresponding namespace URIs. Local mapping might look like this:

File A:

```
{
     "co": "http://example.com/schema1",
     "bdt": "http://example.com/schema3"
}
```

File B:

```
{
     "co": "http://example.com/schema2",
     "wtc": "http://example.com/schema3"
}
```

Listing 6. HashMap for namespace prefixes and their corresponding URIs

The conflicting element definitions and the ambiguities in the schema definitions were discovered at a very late stage in development. If unit tests were written for these cases, these conflicts would have been discovered earlier.

No unit tests were conducted during the development process due to the complexity of the required test setup. Specifically, testing would have required creating multiple interdependent XSD test files, which would have been time-consuming. Additionally, at the time, the structure of the data used in the final *ClimateForest* project was not yet fully defined, further complicating the creation of meaningful test cases.

However, if the XSD parser was to be further developed, future improvements would include:

- Global tracking of Fully Qualified Names (FQNs).
- Locally managing namespace URIs with HashMaps.
- Implementing unit tests that test interdependent XSD files to ensure handling of schemas that contain references.

Finally, as a shortcut, existing crates developed by others to generate structs directly from XSD files were explored. One such crate, *hifa_xml_schema_derive*, was integrated into the project; however, it turned out to be incompatible with interdependent XSD files that include imports. Another option, *xsd-parser-rs*, also generates structs from XSD files but is still under development and has not yet been published as a crate, making it unsuitable for direct integration into the project at this stage.

6 PUBLISHING A CRATE

Of the three projects developed, the second project was the best one. It successfully generates structs from an XML file, making it a practical library for handling individual XML datasets. While the generated structs are specific to a single file, they function reliably within that scope. Because no bugs were identified in the struct generation process, the project was published as a crate.

Crates are open-source Rust projects that anyone can use and contribute to. Once a crate is published, it becomes accessible on crates.io, with its documentation available on docs.rs. Since crates are inherently open-source, my repository's GitHub address is included in the published crate, allowing users to fork or clone the project for their own use. While modifications can only be made locally, users can adapt *schema_generator* for different types of data beyond forestry-related XML. As stated in the README file:

"While it was originally designed for forestry-related XML data, it can be used with any XML data structure, making it a versatile tool for developers working with XML. Additionally, schema_generator supports JSON to XML conversion."

6.1 Steps to Take in Publishing a Crate

To publish a crate, you need to register on crates.io and generate an API token from your account settings. This token is then used to authenticate your local machine by running the following command in your terminal:

cargo login <my-api-token>

The Cargo.toml file is updated to include the appropriate metadata for the crate. This file serves as the configuration for the project, specifying essential details such as the crate name, version, authors, description, and dependencies.

```
[package]
name = "schema_generator"
version = "0.2.8"
authors = ["Marc Bäckman"]
license = "MIT"
description = "This crate is a part of the ClimateForest project,
which takes place between 01.03.2024–28.02.2027. It is funded by
Interreg Aurora and supported by its contributors, including
Metsäkeskus, Lapland University of Applied Sciences, Luke
luonnonvarakeskus, Luleå Tekniska Universitet, Skoggstyrelsen,
and SLU."
repository = "https://github.com/mabackma/schema_generator"
keywords = ["xml", "structs", "json", "parser"]
categories = ["parsing", "data-structures"]
edition = "2021"

[dependencies]
quick-xml = { version = "0.37.0", features = ["serialize"] }
serde = { version = "1.0.207", features = ["derive"] }
serde_json = "1.0.124"
serde-xml-rs = "0.6.0"
regex = "1.11.1"
toml = "0.8.19"
once_cell = "1.20.2"
```

Listing 7. Cargo.toml for schema_generater

Finally, the crate was published to crates.io with the command

cargo publish

6.2 Documenting The Crate

All public functions in the project appear in the published crate, so they must be properly documented. Most of the public functions in *schema_generator* are simple string utility functions that are widely used in larger functions. They were documented with a short comment explaining their purpose. Documentation comments are written similarly to regular comments but use three forward slashes (///) instead of two (//). This ensures that the documentation is accessible online and integrated into Rust's standard documentation system.

```rust
/// Capitalizes the first letter of a word.
pub fn capitalize_word(word: &str) -> String {
    let mut chars: Chars<'_> = word.chars();
    match chars.next() {
        None => String::new(),
        Some(f: char) => f.to_uppercase().collect::<String>() + chars.as_str(),
    }
}
```

Figure 15. Documentation by using comments

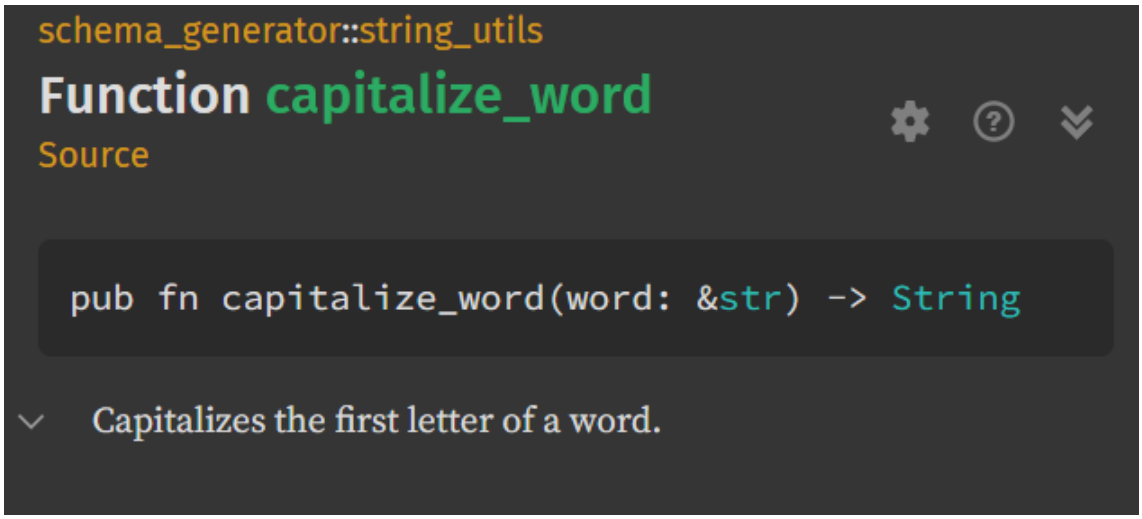

Figure 16. How the documentation looks in the published crate

The two primary functions that handle the core functionality of the project are XML to Struct conversion (**create_structs**) and JSON to XML conversion (**json_to_xml**). These functions were extensively documented, including detailed explanations and usage examples. The examples not only illustrate practical applications of the functions but also provide expected test outputs, allowing users to compare them with their own implementations. The documentation for these two functions can be accessed from these links:

create_structs at *Docs.rs*
json_to_xml at *Docs.rs*

6.3 Tests Inside Documentation

Tests included in the documentation can be executed using the command:

cargo test

A test for the **json_keys_to_lowercase** function is included in the documentation to highlight its utility. This function ensures that JSON keys containing **@** are replaced with **__** after serialization with Serde.

Some tools and systems may not handle **@** symbols in JSON keys correctly, potentially causing errors or unexpected behavior. By substituting **@** with **__**, the function improves compatibility and prevents issues when working with JSON data.

A project called *forestry_structs* was developed for the integration testing of the *schema_generator* crate. It showcases the process of converting XML files to JSON and back to XML using the *schema_generator* crate. The project is available on GitHub and is also listed on crates.io to help users understand how to use the *schema_generator* crate.

7 INTEGRATION TO THE CLIMATEFOREST APPLICATION

When using *schema_generator*, all non-complex field types default to **Option<String>**. Optional types are useful because the XML might contain elements that are missing some fields. However, using *String* for typing is not practical in the forestry application under development.

The application must process large datasets efficiently, requiring lightweight data types to optimize performance. Instead of using **String**, primitive types such as **u8**, **u16**, **u32**, **i16**, **i32**, and **f64** should be used. Primitive types have a fixed size and are stored on the stack, making them significantly faster than String, which involves heap allocation and is relatively slow.

Most values in the XML data are numeric, so to ensure compatibility with the forestry application, struct fields should be typed using primitive types like **i64** and **f64**. An initial approach involved introducing a **HashMap** to store field names and their corresponding types. However, this caused the parser to panic (Rust's reaction to a runtime error) due to inconsistencies in the XML data. Specifically, many floating-point values were represented as 0 instead of 0.0, leading to type mismatches.

To handle the mix of floating-point and integer values, the issue was resolved by using an enum:

```
/// Represents a number that can be an integer or a float.
#[derive(Serialize, Debug)]
#[serde(untagged)]
pub enum Number {
    Int(i64),
    Float(f64),
}
```

Listing 8. Enum used for mixed numeric types.

A custom from_str() implementation was written for the *Number* enum. It converts a String value into an enum variant like **Number::Int(i64)** or **Number::Float(f64).**

To handle deserialization properly, a custom deserializer was written for **Option<Number>**, allowing the enum to be deserialized correctly from optional field types. The custom deserializer is applied whenever the *serde* attributes for a field explicitly reference it.

```rust
#[derive(Serialize, Deserialize, Debug)]
3 implementations
pub struct StIdentifier {
    #[serde(rename = "$text", skip_serializing_if = "Option::is_none")]
    pub text: Option<String>,
    #[serde(rename = "IdentifierType", deserialize_with = "deserialize_optional_number", skip_serializing_if = "Option::is_none", default)]
    pub co_identifier_type: Option<Number>,
    #[serde(rename = "IdentifierValue", skip_serializing_if = "Option::is_none")]
    pub co_identifier_value: Option<String>,
}
```

Figure 17. Serde attribute with custom deserializer

In the example above, the field *co_identifier_type* in the *StIdentifier* struct will use the custom deserializer **deserialize_optional_number** for the optional *Number* type. The attribute **skip_serializing_if = "Option::is_none"** ensures that the field will be skipped during serialization if its value is absent (i.e., None). The **default** attribute ensures that if the optional value is missing during deserialization, it will default to **None**.

The *Number* enum in the *ClimateForest* application enables the use of various numeric types. Functions within the application rely on primitive types like u8 for ID values and u16, u32, and f32 for other data, as using smaller primitive types improves performance. The *Number* enum includes methods to convert between different numeric types for **u8**, **u16**, **u32**, **i16**, **i32**, **i64**, **f32**, and **f64**. The enum and its associated functions are available for review on GitHub.

8 FUTURE WORK

In the future, the *schema_generator* crate could be transformed into an **arena-based** parser. In traditional memory allocation, memory is allocated individually for each specific piece of data or object as needed. Each object must be managed separately, and you are responsible for manually freeing the memory once it is no longer required. In contrast, **arena-based** memory allocation reserves a large contiguous block, known as an "arena," rather than allocating memory separately for each individual object. Multiple objects or nodes are allocated within this block. When the arena is no longer needed, all the memory allocated from it is freed at once. This approach can significantly improve parsing performance and efficiency, especially when dealing with large and complex XML files. (Bylich, n.d.)

9 CONCLUSION

In this project, an XML parser named *schema_generator* was developed specifically for the ClimateForest project. It efficiently generates Rust data structures from XML files using the serde library. After thorough testing, the *schema_generator* crate was successfully published as a reusable library, making it available for other developers to integrate into their applications. While initially designed for forestry-related data, the crate's functionality extends beyond that use case, offering a versatile solution for general XML-to-struct transformations as well as JSON-to-XML conversions.

While *schema_generator* successfully converts XML to structs, another project that attempted generating XSD to structs was not completed. Managing interdependent XSD files and handling conflicting element definitions introduced significant challenges. The creation of unit tests involving multiple XSD files would have been time-consuming, and because the data used in the final ClimateForest was not yet known, the XSD project remains unfinished. The groundwork for generating structs from XSD has been covered and the project could be revisited in the future.

Despite the unfinished XSD functionality, the *schema_generator* crate remains available to developers working with XML data at *crates.io*.

BIBLIOGRAPHY

Bylich, I. (n.d.). *What is an Arena Allocator?* Accessed February 11, 2025, from https://iliabylich.github.io/arena-based-parsers/what_is_arena_allocator.html.

Lapland University of Applied Sciences. (n.d.). Climate change adaptation of northern forests – Risks and prevention of damaging agents. Referenced January 27, 2025, from https://lapinamk.fi/hanke/climate-change-adaptation-of-northern-forests-risks-and-prevention-of-damaging-agents/

OpenGenus. (n.d.). Kahn's Algorithm for Topological Sort. Referenced January 23, 2025, from https://iq.opengenus.org/kahns-algorithm-topological-sort/

Rust Team. (n.d.). The Rust programming language. Referenced January 27, 2025, from https://doc.rust-lang.org/book/

StanForD 2010. Modern Communication with Forest Machines. Referenced January 27, 2025, from https://www.skogforsk.se/english/projects/stanford/

Weik, Martin H. (2000). Computer Science and Communications Dictionary. Volume 1. Springer. p. 662. ISBN 978-0-7923-8425-0.

Wikipedia 1. (n.d.). Geography Markup Language. In Wikipedia. Referenced January 27, 2025, from https://en.wikipedia.org/wiki/Geography_Markup_Language

Wikipedia 2. (n.d.). EPSG Geodetic Parameter Dataset. In Wikipedia. Referenced January 27, 2025, from https://en.wikipedia.org/wiki/EPSG_Geodetic_Parameter_Dataset

Wikipedia 3. (n.d.). World Geodetic System. In Wikipedia. Referenced January 27, 2025, from https://en.wikipedia.org/wiki/World_Geodetic_System

Wikipedia 4. (n.d.). Rust (programming language). In Wikipedia. Referenced January 27, 2025, from https://en.wikipedia.org/wiki/Rust_(programming_language)