Automating XML parser generation for specific data operations

Bachelor Thesis

Latest update: July 6, 2022

Version: 1.13.30

Lars Andringa (S3941906)

Abstract

This Bachelor's project researched whether the generation of XML parsers could be automated if the data operation performed by the parsers were the same for all formats. It found that this is a favourable approach if the amount of parsers that need to be generated was sufficiently high. A proof-of-concept parser generator was developed to research this approach, which demonstrated a high level of accuracy with acceptable levels of performance. Besides generating new tools for the parser generator, other existing tools used during development were analysed and upgraded.

First Academic Supervisor:
Dr. Fadi Mohsen

Second Academic Supervisor:
Professor Jorge Perez

Company Supervisor:
MSc. George Schut

university of groningen / faculty of science and engineering

Schut

Contents

1 Introduction **4**
1.1 Research goals . . . 5
1.2 Research questions . . . 5
1.3 Document layout . . . 6
1.4 Scientific integrity . . . 6
1.5 Acknowledgements . . . 7

2 Background **9**
2.1 Summary . . . 9
2.2 Extensible Markup Language . . . 9
2.3 XML Schema Definition . . . 10
2.4 Regular expressions . . . 11
2.5 State Machines . . . 13
2.6 Context-free grammars . . . 14
2.7 BNF . . . 14
2.8 Parser . . . 15
2.9 Schut array . . . 17
2.10 UBNF . . . 17

3 Related work **19**
3.1 Summary . . . 19
3.2 XML Parsing . . . 19
3.3 Further research . . . 19
3.4 Grammars . . . 19
3.5 Parsers . . . 19
3.6 XSD interpretation . . . 20

4 Architecture **21**
4.1 Summary . . . 21
4.2 Pipeline . . . 21
4.2.1 Frontend/Backend architecture . . . 21
4.3 Two approaches . . . 22
4.3.1 Manual . . . 23
4.3.2 Automated . . . 25
4.4 Technology stack . . . 27

5 Parser building **29**
5.1 Summary . . . 29
5.2 State machine . . . 29
5.2.1 Terminal state machine . . . 31
5.2.2 Grammar state machine . . . 31
5.3 Improvements . . . 32
5.3.1 Error handling . . . 32
5.3.2 Tokenisation . . . 33
5.3.3 Encodings . . . 35
5.3.4 Static Content . . . 36
5.3.5 Grammar parameters . . . 36

5.3.6 Parameter extension . . . 37

6 XSD parsing **38**
6.1 Summary . . . 38
6.2 Introduction . . . 38
6.3 XSD hierarchy . . . 38
6.4 Pre-processor . . . 44
6.5 Post-processor . . . 44
6.6 XSD defined in UBNF . . . 45
6.7 State machines . . . 46
6.8 Output . . . 46

7 XSD conversion **49**
7.1 Summary . . . 49
7.2 Conversion process . . . 49
7.2.1 Constraints . . . 51
7.2.2 Substitution . . . 52
7.3 Class structure . . . 53
7.4 Post-processor . . . 54
7.4.1 Element types . . . 55
7.4.2 Common references . . . 55
7.5 Output . . . 56

8 Analysis **57**
8.1 Summary . . . 57
8.2 Toolset limitations . . . 57
8.2.1 Parser builder . . . 57
8.2.2 Generated parsers . . . 58
8.2.3 UBNF . . . 62
8.3 Performance and accuracy . . . 64
8.3.1 Test setup . . . 64
8.3.2 Performance . . . 66
8.3.3 Accuracy . . . 73

9 Discussion **78**
9.1 Summary . . . 78
9.2 Conclusion . . . 78
9.3 Reflection . . . 79

10 Future work **80**
10.1 Summary . . . 80
10.2 Future work for the company . . . 80
10.3 Future work for researchers . . . 80

11 Glossary **82**

12 References **84**

13 Changelog **87**

A XSD grammar **91**

B Example XSD **99**

C UBNF of example XSD **106**

D Individual results of performance testing **113**

1 Introduction

Extensible Markup Language (XML) is a widely adopted standard for data representation and communication [1]. It provides a wide range of applications in the fields of data storage, data transfer, and software compatibility. Such a wide range of applications would lead many to conclude that XML is a critical part of modern IT infrastructure. Using XML smartly and efficiently can benefit a company's economic performance [2]. The core of these XML technologies are parsers, systems that can read and process XML files. However, developing XML parsers can take a long time, leading to wasted resources and long delays between conception and implementation [3].

The core of any parser, including XML parsers, is a structure referred to as a state machine. Such state machines are essentially a collection of states interconnected through state transitions. These state transitions describe to which states the machine can currently travel and which element should be contained within the input to travel to this new state. The machine then reads the input, and transitions through the states based on the state transitions, to eventually end up either accepting input or rejecting it. In this case, the input does not follow the specification provided by the parser. Section 2.5 provides further explanation and examples. The architecture of this state machine can be defined through a grammar, which is a metaprogramming language. Further explanation, including examples, is provided in section 2.6.

As with any software development field, the dream is a high level of automation. A higher level of automation can either lead to finishing XML projects more quickly or to more complex XML projects being undertaken that were not possible before due to the high development time. It also allows companies to spend more time on other projects, which notably affects revenue. Though there is no scientific classification of different levels of XML automation, my personal view is that there are four levels of automation within XML parser generation:

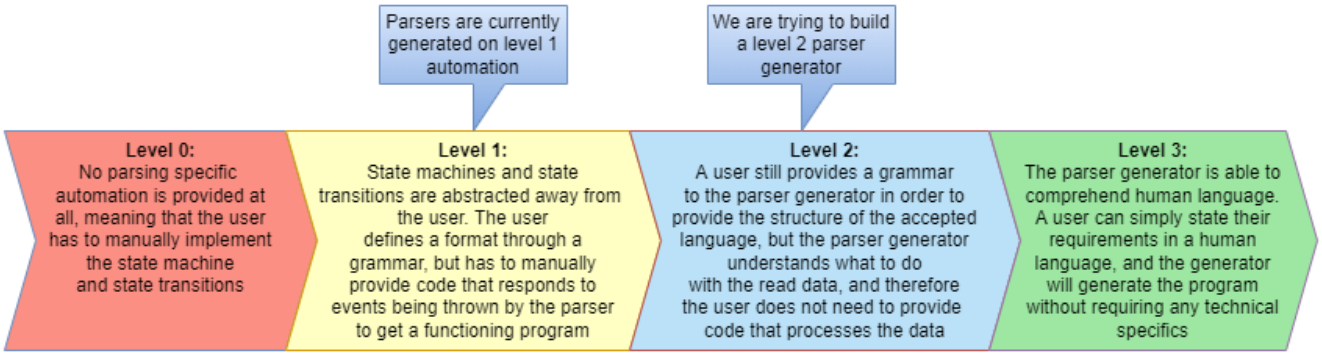

Figure 1: The four levels of parser generation automation.

This project aims to create a parser generator that automatically implements data processing. A user may still write code that works with the output data of the generated parser, but they should not have to write any reactive code to create the parser. The only remaining task in developing a parser is providing a grammar of the language the parser should accept. Designing such a generator is, however, far from trivial. The main roadblock is that a generator cannot know what to do with the data. It highly depends on the application of the parser. XML can be used to describe 3D models which need to be rendered, data that needs to be imported into databases, or content that needs to be formatted and displayed, to give a few examples. This wide range of applications means that a general purpose level 2 automated XML parser is challenging, with impossible not being an unreasonable classification of the problem. As such, specificity is needed.

Therefore we add an assumption to the tool we wish to develop for this project. The assumption states that the specific data operation performed on the XML file remains the same across all formats for which a parser is generated. This way, a tool is created that can automatically create parsers for various formats with the same goal.

This is where Schut B.V. comes into play. The company reached out, requesting exactly such a parser generator, with the goal of generating parsers that can import data contained within XML files into their data structures, referred to within the company as "arrays". Due to the apparent confusion between these arrays and standard programming language arrays, these will be referred to as "Schut arrays". Essentially, a user should provide an XML format, encoded in XML Schema Definition (XSD), and the result should be a parser that can import the XML format into a Schut array. For this project, many internal Schut tools will be used, including a custom grammar language, referred to as UBNF, and custom table builders. These will be treated in future sections.

While multiple tools need to be written from scratch, the company employs numerous tools that are very useful for this parser generator. An effective product will combine the company's tools and personally created tools to generate the final parser. These company tools are, however, highly experimental yet surprisingly innovative. As such, this research will also dive deeply into the toolset to understand, maintain, and upgrade them. As these tools are experimental, they will require upgrading both to support the goals of this project correctly and for future endeavours.

1.1 Research goals

This Thesis will have three goals, outlined as follows:

1. Provide Schut with an effective tool to automate the generation of XML parsers.
2. Research the effectiveness of using a data operation-specific parser generator versus writing a parser from scratch for further usage by researchers and companies.
3. Research the company's unique toolset for parser building, including its limitations, to point out a set of improvements that should be performed.

1.2 Research questions

The following research questions will be investigated and answered during the research:

1. What is the difference in development time between the state-of-the-art approach and the novel approach discussed in this Thesis?
2. What are the performance characteristics of the novel tool discussed in this Thesis?
3. How does the accuracy of this novel tool compare to other XSD tools?
4. What are the limitations of the novel approach discussed in this Thesis?
5. What type of problems can be solved more quickly using this novel approach?
6. What are the technical limitations of the company's tools?

These questions are posed in support of the following main research question:

Could a custom parser generator be a more effective solution for building a set of XML parsers performing the same data operation compared to traditional methods?

1.3 Document layout

The document will start with background information on the topic. Section 2 will discuss background information and necessary theory to understand the Thesis. Section 3 will then discuss work related to this Bachelor's Thesis and the field's state-of-the-art.

Once the background information is closed, the document will go into the architecture of the parser generator and its functionality. Section 4 will give a general overview of the design of the pipeline, which stages it contains, and different iterations of the design. Section 5 will describe the third stage of the pipeline, which involves a significant amount of company tools and their inner workings, while sections 6 and 7 will describe the first and second stages of the pipeline, respectively.

Then next up, the research section of the project is discussed, with section 8 giving an analysis of the tool, its comparison to other tools, and research on the company's toolset. Section 9 will discuss how the research went and how valid the results are, while section 10 will describe work that other researchers in the future can perform.

Finally, a set of miscellaneous sections are to be found, with section 11 covering the glossary of the project, section 12 citing all sources, and section 13 providing a changelog of the document throughout the research period.

1.4 Scientific integrity

Since this project is done in collaboration with a company, it is harder to track the amount of work performed during the project. Furthermore, since some tools will be written by the author, and some are written by the company, it is unfortunately easy to misinterpret sections of the Thesis, which makes it seem like something was the author's achievement, even though the company created it. To ensure absolute integrity, this section will give an overview of all tasks performed by the author during the project and explicitly mention the tasks not completed during the project to ensure there is no confusion regarding the work performed that can arise later in the document.

Some of the items discussed may not yet be known to the reader and will be further described later in the document. This section is purely meant as a reference point for work accomplished and in no way is expected to be understood by someone reading the paper for the first time.

Tasks accomplished during the project:

- First stage: Full development of the grammar describing the parser and syntax that forms the Schut array produced by the parser.
- First stage: A pre-processor that morphs the input files into a more parsable format, and handles the include/import statements.
- Second stage: Full development of the C++ library, which converts the XSD contained within the Schut array to UBNF.
- Second stage: A post-processor that handles references made throughout the XSD that cannot be interpreted by passing over it once using the converter.
- Improved messages produced by the error handler to add more information for the user and make the cause of the parser rejection more clear.
- Improvements to the tokeniser such that it makes smarter decisions about which token to retrieve next from the text.
- Additions to UBNF to allow a user to define static content to be added to the output Schut array.

- Improvements to the parser generator to optimise parameters, which reduces memory usage and file size of produced parsers.
- Improvements to the text decoder of the parser builder to allow more encoding formats to be used for files being imported into the parser builder.
- Changes to the GUI and underlying codebase for the parser builder to integrate the new parsing option (XSD) developed during the project into the parser builder on top of existing parsing options (UBNF).
- A standalone program that can run the second stage of the pipeline, which was initially used to test the entire pipeline and develop the second stage manually.

Tasks not performed by the author:

- Development of the UBNF grammar (except for changes mentioned above).
- Development of the parser builder (except for changes mentioned above).
- Development of the parser generator (except for changes mentioned above).
- Development of the Schut array.

Project statistics:

- Size of the first stage (XSD parser grammar): See appendix A
- Size of the second stage (converter):
  - First attempt (Now fully archived): 571 lines of C++ code
  - Final product: 1890 lines of C++ code
  - The first attempt shares no common code with the final product.
- Third stage (parser builder):
  - Lines of code modified: Approximately 1000, it was not possible to perfectly measure as the company's version control did not support lines of code modified, and the files are spread out throughout the codebase.
  - Note that the generation of the state machines is a highly complex process, so significant time has to be spent analysing and understanding the surrounding infrastructure before any modifications can be made.
- Estimated hours spent on Thesis writing: 120
- Estimated hours spent on development: 200
- Estimated hours spent on analysis and research: 80

1.5 Acknowledgements

I would like to thank the following list of persons for helping with the research and writing of the Thesis.

- George Schut: For the immense support as the company's supervisor during the project. George provided all the resources possibly needed to complete the project, spent time helping to navigate the company's codebase, and provided feedback on both the software design and the Thesis.

- Fadi Mohsen: For being the academic point of contact for the project. Dr. Mohsen helped tremendously with the research part of the project, giving great feedback on issues within the document's structure, pointing out mistakes, and giving recommendations on the research part of the project
- Jorge Pérez: For being an easily reachable second supervisor. As a second supervisor, professor Pérez's responsibilities were limited, but he was always readily available and reachable in case anything was required, and spent time meeting and giving advice.

2 Background

2.1 Summary

This section will concern itself with the background knowledge required to understand the bachelor project. Its priority is to give the reader a basic understanding of the core concepts of parsing, and a simple introduction into the internal tools of Schut which are relevant to this project. It will cover the basic parsing concepts of regular expressions, state machines, grammars, and parsers themselves, while it will also go over relevant technologies like BNF, UBNF, Extensible Markup Language, XML Schema definition and Schut arrays.

2.2 Extensible Markup Language

Extensible Markup Language (XML) is a text format specifciation to produce text that is both humanly readable, and machine-readable. It was derived from SGML [4]. An XML document consists out of tags. A tag is surrounded by a less-than sign and a greater-than sign. A tag can either be a single element, described as "<element/>", or can occupy a region of the document, where it consists out of an opening and closing tag: "<element> somethingElse </element>".

Each element can contain attributes, content, and children. An attribute is described in the opening tag, like "<element name="first"/>", content is provided within the region between the tags: "<int> 5 </int>", and children are simply tags contained within other tags: "<array> <int/> <int/> <int/> </array>". With this, one can create a hierarchical structure that stores any data one may wish. This can create complex data structures, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>Light Belgian waffles covered with strawberries and whipped cream</description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>Light Belgian waffles covered with an assortment of fresh berries and whipped cream</description>
    <calories>900</calories>
  </food>
</breakfast_menu>
```

Figure 2: Example retrieved from https://www.w3schools.com/xml/simple.xml

Essentially any file that keeps to this structure is a valid XML file. The specific keywords used within the tags is not relevant to the validity of the XML file. As such, users can define their own data format by specifying the names of elements, and which elements can be children of an element. Beyond this, a user can also specify how

many children an element can have, and whether certain children are mandatory. A programmer can then create tools that check whether an XML file follows the structure that the programmer expects, and then process the file.

Whenever a file is provided that either is not valid XML, or does not follow the exact specification that the programmer expects, the XML file is said to be invalid, and is rejected by any tools trying to process it. This can either be done by first checking whether the file is valid, and then going over it a second time processing it, or checking for validity during processing, and stopping when an unexpected element is encountered (or an element is unexpectedly missing).

2.3 XML Schema Definition

XML Schema Definition (XSD) provides a standardised manner to define XML formats. It allows a programmer to specify the names of elements within the format, which elements can be children of which other elements, and what attributes they have. It also allows one to specify lots of smaller details, like how many times an element is allowed to appear as a child, whether there is any sequence to the elements appearing, the datatype of the attributes, and many other features.

XSD itself is an XML format. This means one actually would write an XML file which describes how another XML file should be formatted [5]. The core of this schema is the "<xs:element>" tag. This specifies a tag within the schema. One needs to add a name attribute to it, which specifies how the tag is named. This element can then have child elements, like other xs:elements, or xs:attributes, which specifies an attribute that the tag has.

While xs:attribute can be a direct child of xs:element, which then specifies an attribute that the xs:element contains, an xs:element is not a direct child of another xs:element. As the actual structure of describing a child element of another element is quite complex, an example will first be provided:

```
<xs:element name="shipto">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
</xs:element>
```

Figure 3: Example adapted from https://www.w3schools.com/xml/schema_example.asp

Note how an xs:atribute is a direct child of an xs:element. To provide child elements, we must first specify "xs:complexType" as a child to xs:element. Doing this specifies that shipto is an element that contains other elements as children. The alternative would be "xs:simpleType", which would contain no child elements, but can still specify things such as restrictions on the element.

The complexType then contains something called an indicator. In this case, the indicator is xs:sequence. The sequence specifies the order of the elements provided within the indicator. xs:sequence means that they have to appear in exactly the order provided in the XSD. Other possibilities exist too, like xs:all and xs:choice, which specifies that the order is irrelevant or that the user is only allowed to choose 1 of the elements to use. One can then finally add children by adding child xs:elements to the indicator.

Figure 4 describes a full XSD example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Figure 4: Example taken from https://www.w3schools.com/xml/schema_example.asp

2.4 Regular expressions

Working with regular expressions is a critical skill within parser development. A regular expression allows one to specify a pattern that matches a set of words. Essentially one can specify a property that a word should possess, and then all words that match that property are accepted if compared to the regular expression. While it is not immediately clear what "a property" means, for now, it is sufficient to understand that a regular expression allows one to check whether a word matches a particular property.

Regular expressions can be used to perform tasks such as:

- Find all words that match the regular expression in a piece of text.
- Check whether a specific word matches a certain property (is it a phone number?).

- Split a piece of text up into a list of tokens, where each token is assembled from characters based on the list of regular expressions provided.

This third task is of specific interest to us. Parsers work through tokens, which inherently means that a piece of text is split up into its primitives, where each primitive is a word with some meaning. The following example shows such a tokenisation of a piece of text.

```
<Element>
    <Value val="1"/>
    <Value val="4"/>
</Element>
```

Figure 5: The XML file to be tokenised.

```
< Element >
< Value val = "1" / >
< Value val = "4" / >
< / Element >
```

Figure 6: The list of tokens produced. Tokens are separated by either a whitespace or a newline.

One can then use this list of tokens to parse the file and check whether the file matches the format. One has to specify how to generate the tokens manually; this is where regular expressions come in. You can specify words, like stating that the "/" character is a separate token or that "Element" is its own token. But with regular expressions, you can also give smarter definitions of tokens. Like if you want to take any number (a string of digits) and make it a token, you can specify "[0-9]+", where [0-9] indicates that it should be any character between 0 and 9, and the + indicates that this character should appear between 1 and infinite times in a row, effectively defining a string of digits. Regular expressions can get much more complicated, but this example catches its essence, where you can define an infinite amount of words that would be valid as a token using a single regular expression.

The following table specifies a list of valid tokens within regular expressions:

Character	Brief Description	Simple Example
.	any character (wildcard)	w..kly
^	Begins with	^where
[ ]	character set	[0-9]
|	either or	hi|bye
+	Once or more	me+
*	zero occurances or more	me*
{}	exact number of times	I{4}
\	special character operations	\w
$	Ends with	$bar

Figure 7: Special characters in regular expressions, retrieved from [6]

2.5 State Machines

A state machine is the most popular method used to implement grammar related tools like lexers, parsers and regex implementation [7]. They allow a programmer to set up an architecture closely related to a directed graph (see glossary), that is able to recognise whether a piece of text complies to a specific format. The complexity handled by these architectures can range from simple regular expression matching, to reading in entire programming languages.

The following is an example of a "finite state machine", which is one of the more basic implementations of a state machine:

δ	a	b	c
→ q_0	q_1	q_2	q_3
* q_1	q_2	q_1	q_2
q_2	q_2	q_2	q_2
q_3	q_0	q_2	q_2

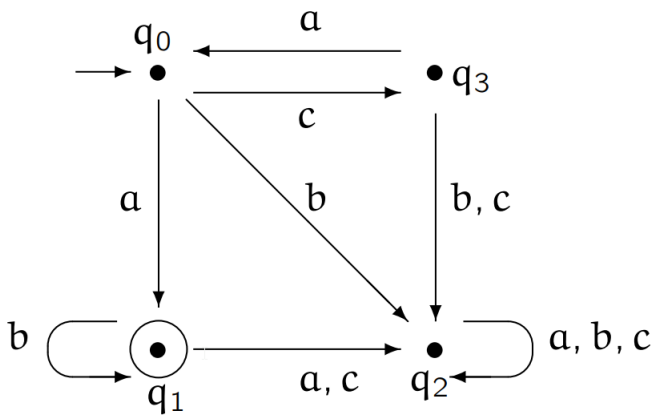

Figure 8: An example of a state machine, retrieved from Languages and Machines lecture notes, example 3.1 [8].

The machine described in figure 2.4 checks whether a string adheres to the regular expression "(a | b)*aaa". Such a machine consists of two essential elements: States and transitions. When comparing it to directed graphs, a state is a node, and a transition is an edge. At any moment in time, we are in some state. From a state, it is defined to which other state you are allowed to travel to and what needs to be in the input string to be able to transition to that state. An example is: "From state 2, you transition to state 4 if you read in a b". We transition through the states while reading in a string based on what we are reading in the input string. A string is considered invalid if one of the two conditions are met:

1. No transition is defined from the current state to another state for the next character in the input string.
2. The entire string has been read in, but we are not in a final state.

The final state requires further explanation. Some states are considered final states, which one is always allowed to enter or exit. A string is however only accepted if the machine is in a final state when the string has been fully read it.

When looking at figure 8, we start in state q0. From q0, the machine can transition to state q1 if it reads in an a, q2 if it reads in a b, or q3 if it reads in a c. From these new states, it can then again be checked which state can be transitioned to using which input characters. The table to the left of the figure shows which states can be transitioned too.

Let us imagine we have the string "caab", when looking at our state machine, the transitions go as follows:
$q0 \rightarrow q3 \rightarrow q0 \rightarrow q1 \rightarrow q1$
As we end in q1, our only final state, the string is accepted.

When we however provide the string "bac", the following transitions happen:
$q0 \rightarrow q2 \rightarrow q2 \rightarrow q2$
As q2 is not a final state, the string is rejected.

Finite State Machines are the simplest of state machines. They are quite limited, and often purely reserved for regular expressions, where it is powerful enough to match any regular expression imaginable. More complex machines make use of stacks to remember information like previous read in characters on top of their state (push-down machines), while other machines even have random-access storage to use while reading in information (Turing machines) [8]. These are, however, outside the scope of this Thesis, as parsers do use very complex state machines, but maintain different implementations to achieve this complexity.

2.6 Context-free grammars

A context-free grammar is essentially similar to a regular expression, in that you can describe the structure of a string. However, it is more potent than a regular expression, with its main advantage being that it can "remember" earlier parts of the parsed string.

An example of a string that a context-free grammar can parse but a regular expression cannot is the string $a^i b^i$ (A certain amount of a's followed by a certain amount of b's). With a regular expression this is not possible, as the underlying algorithm is not able to remember how many a's it has encountered by the time it gets to the b's.

Context-free grammars work through something referred to as production rules. A production rule is a rule that can recursively be applied while parsing a string. For $a^i b^i$, the following production rule would parse it:

$S \rightarrow aSb \mid \epsilon$

How you read this is that when you start parsing a string, you start with the root symbol, in this case, S. You then repeatedly apply the production rules until you reach the string you are trying to match. So essentially, if one were to parse "aaabbb", it would be parsed as follows:

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$

Note that we can either replace S by aSb, or replace S by ϵ (ϵ means it is replaced by absolutely nothing, the empty string). Within a production rule | splits the different options one can take. One can also define multiple production rules. For example, if we had the string $a^i b^j c^j d^i$, we could use the following grammar:

$S \rightarrow aSd \mid G$
$G \rightarrow bGc \mid \epsilon$

Here the S rule will first set up the a and d characters on the outside of the string, and then the G rule sets up the b and c characters of the string. Note that b and c do not have to be of the same amount as a and d.

2.7 BNF

Backus-Naur form (BNF) is a manner to write a context-free grammar. The difference between the notation used in the previous section and BNF is that BNF is easier to write in a text file, as BNF does not use the $\rightarrow$ character, but instead describes production rules as:

S ::= A | B;

Where ::= equals the → character in context-free grammars. Some implementations expect BNF to have < and > character surrounding the production rules, like:

<S> ::= <A> | <B>;

But this is by no means mandatory, and Bison, a very frequently used implementation of BNF, does not use these surrounding characters.

2.8 Parser

A parser is an algorithm that takes in a piece of text, and then matches the text structure against some known structure built into the parser, like a programming language or data format. It often uses some grammar like BNF to define the format and then uses this grammar to match the text against it. By reading the text and matching it against a structure, the parser can pick out certain elements within the structure. For example, if we look at the XML file defined in figure 5, one can see that the root structure of the file, defined as a regular expression, is:

<Element> (<Value val= number />)+ <Element>

Where, as explained in section 8, the + indicates that the section between brackets can appear multiple times, with it occurring at least once. When we have a piece of software that matches a text precisely against such a structure, we can tell the parser to run some code when it encounters the number, like storing the value. This is the strength of parsers. They not only allow you to match a piece of text against a structure but also allows one to provide code that reacts to a particular part of the known structure being encountered.

A parser essentially uses a state machine to parse the text. Such machines were already described in section 2.5. One thing where parsers differ from the theoretical nature of state machines is that they use recursion and backtracking. When a parser has multiple directions within the state machine that it is allowed to go, a parser can choose to go into a direction, look at what it expects afterward, and if it at some point gets stuck, it can backtrack and try the other route. This is what is referred to as a parser looking ahead. However, such backtracking reduces the performance of the parser. As such, there is often a trade-off for parser design between how far they can look ahead and how efficient they are. The more common parser designs, like LL(1) and LR(1) [9], have minimal look-ahead capability. The parser designed by Schut, however, has a very high look-ahead capability, leading to the more straightforward design of parsers but worse performance.

Another important design choice is the type of parser. There are two common designs, bottom-up parsers and top-down parsers. If we were to take the following BNF grammar:

S ::= A S B | S S | c

A ::= 'a';

B ::= 'b';

Figure 9: Example grammar

And then try to parse the string "aacbbacb" with it, a top-down parser would match it as follows:

$S \rightarrow SS \rightarrow aSbS \rightarrow aaSbbS \rightarrow aacbbS \rightarrow aabbaSb \rightarrow aabbacb$

Essentially it starts with the root grammar rule, and applies the grammar rules until it finds a matching string. Bottom-up parsers work very differently:

$epsilon \rightarrow a \rightarrow aa \rightarrow aac \rightarrow aacb \rightarrow aS \rightarrow aSb \rightarrow S \rightarrow Sa \rightarrow Sac \rightarrow Sacb \rightarrow SS \rightarrow S$

Rather than starting at the root symbol and applying grammar rules, it reads in the string one character at the time, and converts sections of the string that match a grammar rule into that grammar rule. At every iteration, it either appends the next character from the input to the string, or replaces part of the input with a matching grammar rule.

The algorithms that actually run top-down and bottom-up parsers are quite complex, and as such will not be covered. One thing to note however is that top-down parsers are easier to implement, while bottom-up parsers are more flexible and powerful. As such, most parsers nowadays are bottom-up.

When looking at the grammar described in figure 9, the following state machine is generated to parse a file following the grammar:

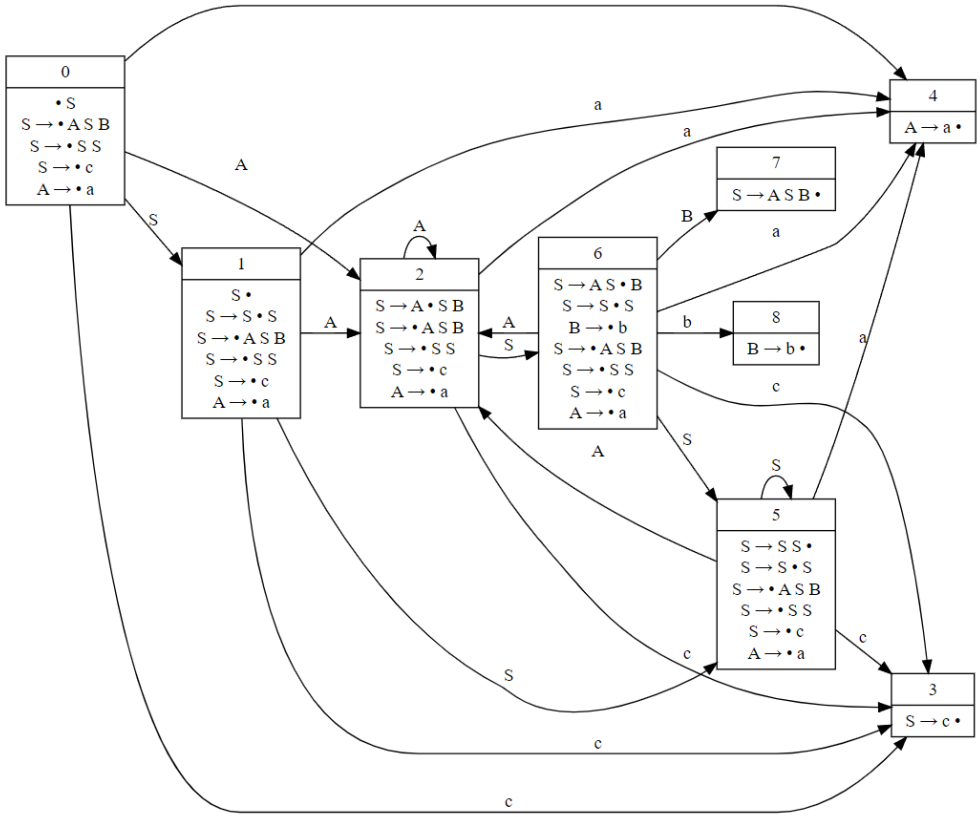

Figure 10: State machine generated using the grammar illustrated in figure 9, generated through https://mdaines.github.io/grammophone

The black dots seen within the grammar rules being treated in a particular state is the separator between the part of the grammar rule that already got read in, and the part of the grammar rule that must still be encountered.

Essentially the dot moves along the rule as we parse the text using the rule. Furthermore, the grammar rules seen within a state are the transitive closure of the states being treated. So if the rule $S \rightarrow ASB$ is being treated, and we are at the start of the rule, then $A \rightarrow a$ is also being treated, as one would need it to know which character is expected next.

2.9 Schut array

The Schut array is one of the core technologies used within this Thesis. It is both a significant part of the destination and of the process. A Schut array allows a programmer to maintain a heterogeneous array, while also allowing the data to be structured in a hierarchical manner.

The starting point of understanding the Schut array is understanding that it is in its core essence just an array. It is an array of elements, for which each element is heterogeneous. This entails that the elements within a Schut array do not need to be of the same data type. Beyond the type of the elements of the Schut array being dynamic, each element is associated with an ID, which may or may not be guaranteed to be unique depending on the parameters of the array. As such, each element within the array contains two values, its identifier, which is of the type wide string (a string that supports a considerable amount of characters), and a value, which type is dynamic and defined when pushed onto the array.

An interesting part of the Schut array is that a Schut array is also a valid value type for its elements. As such, one can define elements within the Schut array to be a Schut array. This allows for hierarchical data ordering, where a tree-like structure is simulated through nesting Schut arrays.

The following diagram shows an example of how a Schut array works:

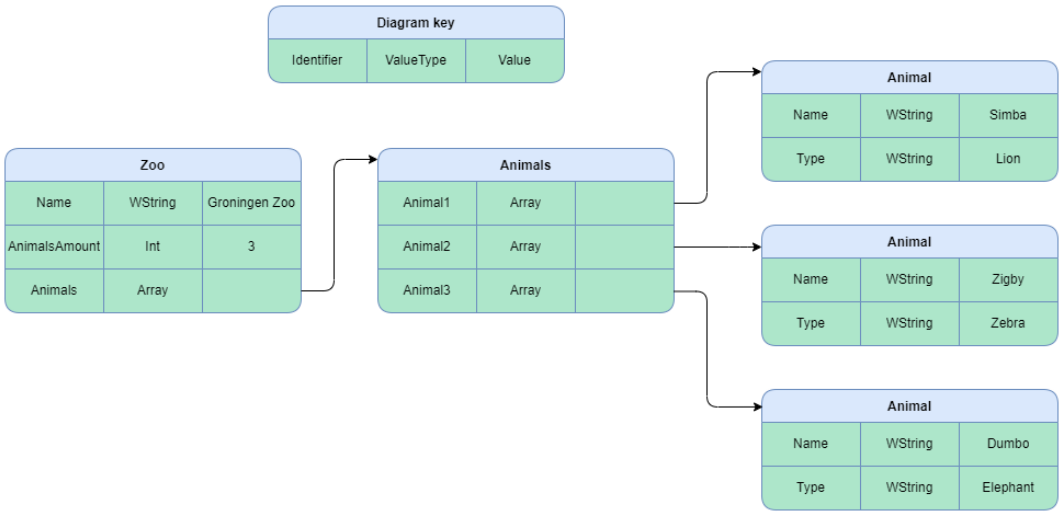

Figure 11: Example structure of a Schut array.

2.10 UBNF

UBNF is an extension of BNF developed by Schut. It adds a large number of new features. These changes can be grouped into two categories.
The first category is the ability to use regular expression features within the grammar. In classic grammars, one would need to write a token that can appear any amount of times in a row as follows:

A ::= token A | ϵ;

Where one can do the same in UBNF using the following syntax:

A :=token+;

Some implementations of BNF could already do this, but BNF is in practice simply a computer-friendly way to describe context-free grammars, which do not support regular expressions within their rules.

Secondly, UBNF contains many grammar features that are used to import the data into a Schut array. The grammar rules are added to a Schut array, where every terminal is a WString within the Schut array, and every non-terminal is a Schut array. The following example shows the UBNF grammar, file it is being applied to, and the resulting array.

```
<S> := <A> <B> <C>;
<A> := ”a” ”a” ”bb”;
<B> := ”c” ”c” ”dd”;
<C> := ”e” ”e” ”ff”;
```

Figure 12: The example grammar in UBNF.

```
aabb
ccdd
eeff
```

Figure 13: The example file parsed by the grammar.

Then the following array would be generated:

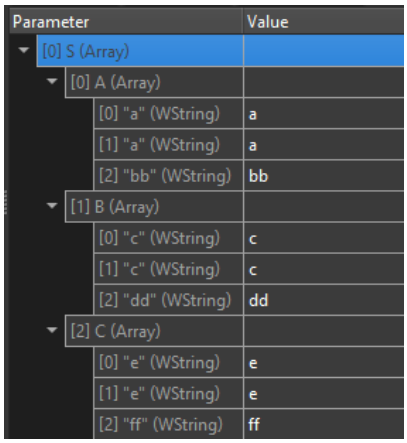

Figure 14: The resulting array from parsing the file using the grammar.

UBNF contains grammar features to modify the resulting array. For example, if one would want A, B and C to be the root elements and S to not appear in the array, one can state:

```
<S; Alias=true> := <A> <B> <C>;
```

Furthermore, UBNF contains features to not add certain elements to the array, either by explicitly stating which elements should be used, or by specifying per token to not include it, and features to rename the identifier of an element.

3 Related work

3.1 Summary

This section will be focused on research into previous work on the topic, in order to discover the current state of the art that needs to be extended upon, and which topics do not require renewed research. It will focus itself on the core topic on XML parsing, but will then also extend to research on related topics that will be heavily used within this project, like grammars, parsers and the XSD format.

3.2 XML Parsing

The state of the art in XML parsing presents itself as an unusual point of research. It is a prodigiously widespread standard [10], yet its research coverage leaves much to be desired. Current research is highly fixated on improving the performance of general-purpose XML parsers, as XML is notoriously slow to parse [11]. Examples of such research are double-lazy parsers [12], direct schema compilation [13], parallelisation [14], and bottom-up mining [15]. Such research mentioned before is mainly focused on solving the inefficiency of DOM-based parsing [16].

Yet this race for the best general-purpose parsing algorithm disregards the development speed aspect of XML. The result is that little progress is made on directly automating XML parser generation. Researchers are attempting to find new ways of generating XML parsers, but the generated programs often do not scale beyond format compliance tests [17][18][19], or require manual input on what to do with the data [20][21]. That said, research has been performed on detecting issues with XML documents that scales beyond simple format noncompliance [22]. The most advanced research in the field of XML parser generation for specific data operations revolves around libraries that use common interfaces to merge different formats. One research paper has shown this to be an effective tool to visualise biomedical data stored as XML [23]. To my knowledge, there are no research examples yet of direct XML parser generation which generates parsers that can parse, process and use data in a larger program without any programmer intervention.

3.3 Further research

While XML parser generation is the main goal of this research, this goal will be achieved through the use of grammars, parsers and XML Schema Definition (XSD) interpretation. These are not trivial topics, and will also require non-traditional solutions. As such, it would be useful to also investigate these topics' state of the art.

3.4 Grammars

With regards to grammar, one commonly investigated topic is extending the semantics of the well-established Backus-Naur Form (BNF) grammar notation to improve readability and maintainability when defining grammars [24][25][26]. A note to make here is that such extensions generally do not increase the expressiveness of BNF, but instead reduce development time in highly idealistic scenarios. One newer area of semantic extensions to BNF seem to be in the field of graphic modelling [25][27]. Yet another type of research in grammars is the automation of conversion between different grammar definitions [28]. Interestingly, We are not aware of any effort to replace BNF as the de-facto grammar definition notation, which is noteworthy, as the initial ideas of BNF stem from 1959 [29]. Most features contained within UBNF seem to not be novel, but rarely used within academic research.

3.5 Parsers

A large field of research within parsers is the efficiency of parsers. LR(1) parsers are considered a great class of parsers, but often require very large state machines. LALR parsers attempt to solve this exact issue, but are less powerful than LR(1) in practice [30]. A significant chunk of parser research is focused on finding a powerful, yet

highly efficient alternative [30]. Furthermore, systems have been developed that convert lower-effiency parser classes to higher-efficiency classes, which allows defining grammars in more accessible parsers, and then converting them to higher-efficiency ones [31][32].

3.6 XSD interpretation

Converting and interpreting XSD contains very little up-to-date research. This is because this is a relatively simple field, with little optimisations to perform, as XSD was specifically meant to be able to be able to be read easily. New methods are however being researched for converting other formats to XML schemas [33][34]. Converting XSD to a grammar like BNF seems to receive little attention, as XSD was intended for XML primitives like elements, attributes and children, rather than tokens [5], and would to my knowledge be a relatively novel concept for this research.

4 Architecture

4.1 Summary

This section describes the general architectural overview of the parser generator tool. The tool will consist out of a set of stages, which are run sequentially. So while future sections will focus on individual components, this section will focus itself on how those components are interconnected and the parser generator's working from a global perspective. It will explain the pipeline, how the pipeline can be constructed, and the two different designs of the pipeline that were created throughout the project.

4.2 Pipeline

The parser generator developed during this project is a 3-stage pipeline. 3 separate programs are run sequentially, with the output of a previous stage being connected to the input of the next stage. The input of the pipeline is an XSD file defining the format of XML files that should be parsed, while the output of the pipeline is the parser, which is in the form of two state machines. These two state machines entirely define the behaviour of the parser, and are discussed later in the Thesis.

The following image describes the pipeline of the parser generator:

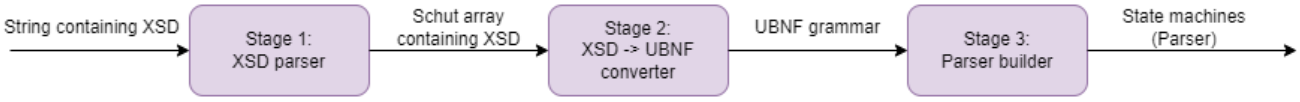

Figure 15: The 3 stages of the pipeline.

The goal of the first stage is very simple: We need the XSD format to be machine readable, so we can utilise it to generate the state machines. Essentially this stage needs to read the content of a file in text-format, and load it into a data-structure that can be iterated over to view the values contained within the XSD. This stage is discussed in section 6.

The goal of the second stage is to use this machine-readable XSD to generate a UBNF grammar from it that both matches the format of the XSD, and contains the necessary syntax for the parsers to be able to export the data from the XML file into a Schut array. This stage is discussed in section 7.

The third and final stage's goal is to use this UBNF grammar generated by the second stage to generate the state machines that define the parser. This system was originally developed as an experiment by Schut, and is as such not fully created by me. We have made improvements and changes to the system, but have not originally designed or developed it. This stage is discussed in section 5.

As such, to conclude, the first and second stage is to be fully built by me, while the third stage is already present in an experimental form, and simply requires modification and upgrading to be usable within the project.

4.2.1 Frontend/Backend architecture

When glancing at the pipeline, it becomes quite clear that there are essentially two steps being taken:

1. Convert the XSD to UBNF
2. Convert the UBNF to the parser

This could be seen as excessively complicated, as one could simply directly convert from XSD to a parser. However, it is in fact a sound design, following the frontend/backend architecture, a frequently used architecture within compiler design [35]. Within this architectural design, a compiler is split into two parts: A frontend and a backend. A frontend would convert from the original programming language to an intermediate language, while the backend would convert from the intermediate language to the final executable.

The reason that such a design is sound, is because it gives compilers a very interesting property: Creating a new compiler only requires one to develop half of the compiler pipeline. This is because for compilers, when creating a new compiler, it is generally different from previous compilers in one of two ways: It either is made for a different programming language, or has a different target architecture. If one has a C compiler for Windows, and wants to make a Fortran compiler for Windows, one simply has to develop a new front-end, and attach it to the same back-end used by the C compiler. If one wants to make a C compiler for Linux while already having one for Windows, the backend is the only part that needs to be swapped out.

Within our pipeline, the exact same principle applies. If one wants to support a different format specification language, which can either be another XML format specification like DTD, or even one for an entirely different type of file format, one simply has to swap out a different front-end. If one wants to however keep the same format, but change what the parsers generated will do with the data, only the back-end has to be changed to make such a change possible. This principle is captured in the following diagram:

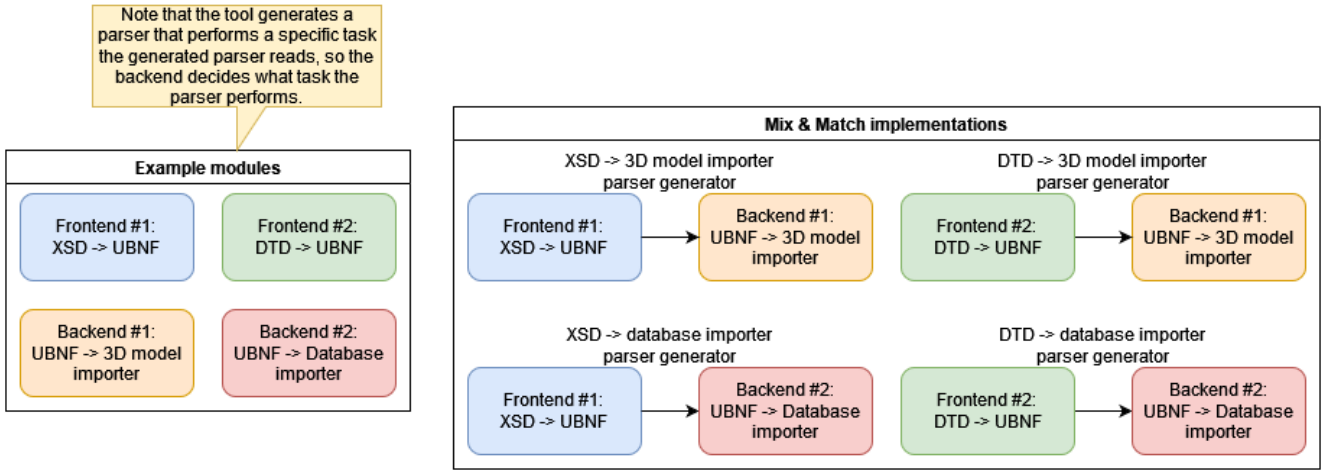

Figure 16: The frontend / backend principle of the parser generator.

4.3 Two approaches

This project is of a very significant size, with many sub-components that are not critical to the completion of the project. As such, risk reduction is an important soft skill for this project. It is critical that the first priority is finishing the core tasks of the project, and then improving quality of user interaction and extra features with the remaining time.

Multiple systems required for this pipeline are already developed by Schut, but require time to integrate into a fully automated pipeline. This mainly includes the Schut array, UBNF language, and UBNF parser builder. Knowing this, the first implementation is a very manual one, where stages are separate applications, that the user can use to create the final product themselves. Essentially, a user is able to create the final product, but needs a high

level of knowledge about the process, and takes some work to transfer between the stages of the pipeline.

Due to this manual approach, current Schut applications can directly be used, together with the applications developed by me, with only a limited amount of time being invested in modifying and adding onto the Schut applications to support my own project. Once my own tools for completing the pipeline are developed, We then create one final application that runs all Schut and personal tools in a pipeline. It is critical that this is performed at a later date, because it takes time to reverse-engineer the Schut applications, and creating an automated pipeline can at best be classified as an "important requirement", and not a critical one. Critical requirements are creating some method to generate the final product, and improving and extending existing Schut applications like the parser builder.

As such, the manual approach is a proof of concept and first draft of a fully working pipeline, whereas the automated approach is a fully flushed out final product.

4.3.1 Manual

When interacting with the manual approach, a user would use two tools. The first is the Schut parser builder, and the second is the UBNF converter. Using the two tools in an intelligent combination allows a user to complete the whole pipeline, even if it requires unnecessary manual work.

The Schut parser builder is a tool developed by Schut that allows parsing and converting files into a Schut array using a UBNF specification. Despite being available at the start of the project, it is a highly experimental tool and does require modifications and validation of the codebase to be properly and safely used. An example of an added feature is a method to export the Schut array generated by the program, which previously only allowed itself to be displayed within the program's GUI. Library features were, however, already available to complete such a task, so it was purely a matter of setting up the correct UI elements and calling the proper library functions.

The syntax-highlighted textbox provides the UBNF grammar describing a file format on the left. The file itself is provided in the top-right, in the non-syntax highlighted textbox. The output Schut array is provided in the box below the top-right textbox. Other areas offer extra tools such as debug and error logs.

Now that we have a tool to parse files using UBNF, which we can use to parse the XSD file, and then parse the final XML file, we only need a tool that can convert the parsed XSD file to UBNF so that we can use this grammar to parse the final XML file. For this purpose, a new tool must be developed to perform this task. This new tool will be called the converter. It is essentially a straightforward UI that asks the user about the file's location to import and the location to put the UBNF file, and then converts it using a rather complex conversion algorithm discussed in later sections.

The following diagram explains the full workflow of the manual pipeline:

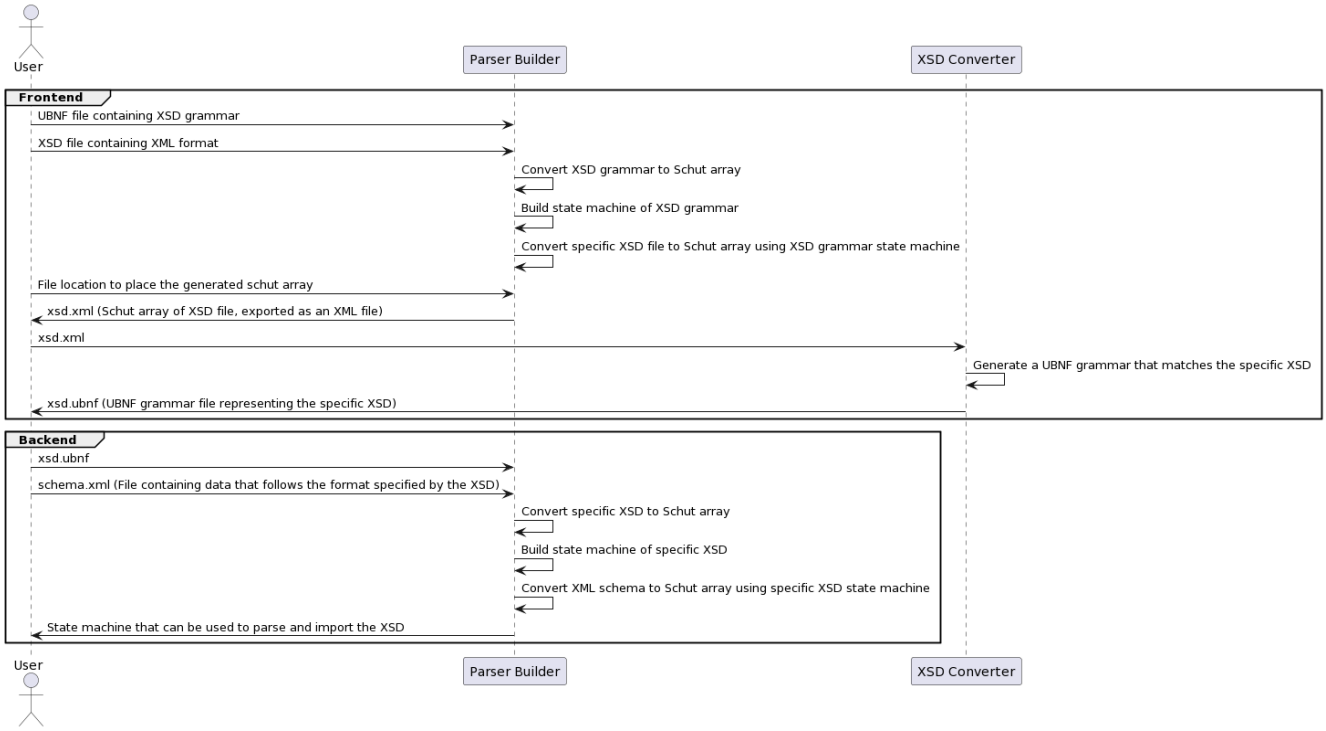

Figure 17: The manual workflow.

As can be seen, the workflow consists of 3 interactions. The user first interacts with the parser builder to generate a Schut array containing the XSD format, making the XSD machine-readable. This machine-readable XSD is then used to create the UBNF grammar that defines the XSD format, and then this grammar is used to generate the final product, a Schut array containing the data from the XML format.

Note that terminology can be very confusing in this pipeline. The main confusion is centred around the definition of an XSD grammar. Within this diagram, the terminology "XSD grammar" directly refers to a grammar defining the notation of XSD (so how an XSD file is structured). In contrast, the terminology "specific XSD" refers to an XML format defined using XSD notation. These are important to distinguish, so care has been taken to use exactly this terminology.

Note that the final XML Schut array is not returned to the user. Returning the Schut array is undoubtedly a possibility, but the user can decide what to do with the Schut array, and exporting is not necessarily the only option. A user may also directly use the Schut array in their own workflow, which may not require exporting the array. Once the Schut array containing the XML data is generated, the pipeline's job is done.

This manual pipeline has three limitations:

1. A significant amount of unnecessary manual labour is required to complete the pipeline. It is perfectly reasonable for the software to be able to generate the final array when it is just provided with the XSD file.

2. The XSD parser is generated on every run, even though this parser will stay the same every time. This inefficiency is because the parser builder was made to develop parsers; therefore, the parser changes typically. It is, however, more sensical to keep the XSD parser as a pre-generated static object.

3. XSD files can include other XSD files in their definition. This feature is tough to support with the current implementation, as it would require the user to convert all XSD files to Schut arrays and then provide all of them to the XSD converter before starting the UBNF conversion. It would be more sensical for the program to parse the XSD file manually and then recursively start the parsing process on all included XSD files. But since the parsing step is manually performed, this is not possible with the current workflow.

The automated pipeline attempts to solve these issues.

4.3.2 Automated

The automated pipeline is a much more intelligent system. It integrates the full pipeline into an automated process that takes no user interaction once it has started. This means a user purely has to provide an XSD file containing the XML structure, and the output is the generated parser.

A possible approach could have been to create a stand-alone application that handles all of this. However, Schut already possesses a tool that deals with importing a file into a Schut array, called the "Parser builder". This tool takes in a UBNF grammar of the file and generates a parser that can parse this grammar and import it into a Schut array. Rather than creating another application, we can set up a system within the tool where the parser builder automatically detects the file type. It then either applies the UBNF parser generator if the imported file is a UBNF file, or applies the XSD parser generator if the file is an XSD file. This means no user interaction is needed to select the correct mode, making interaction with the tool easier. The following images show the tool in the different modes:

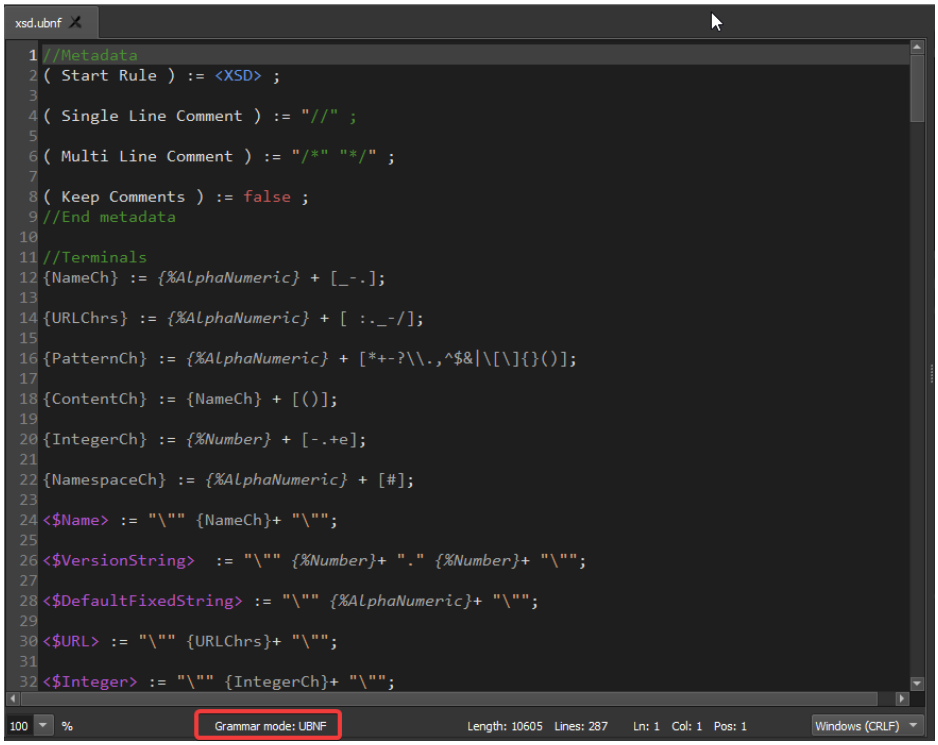

Figure 18: The parser builder in UBNF mode

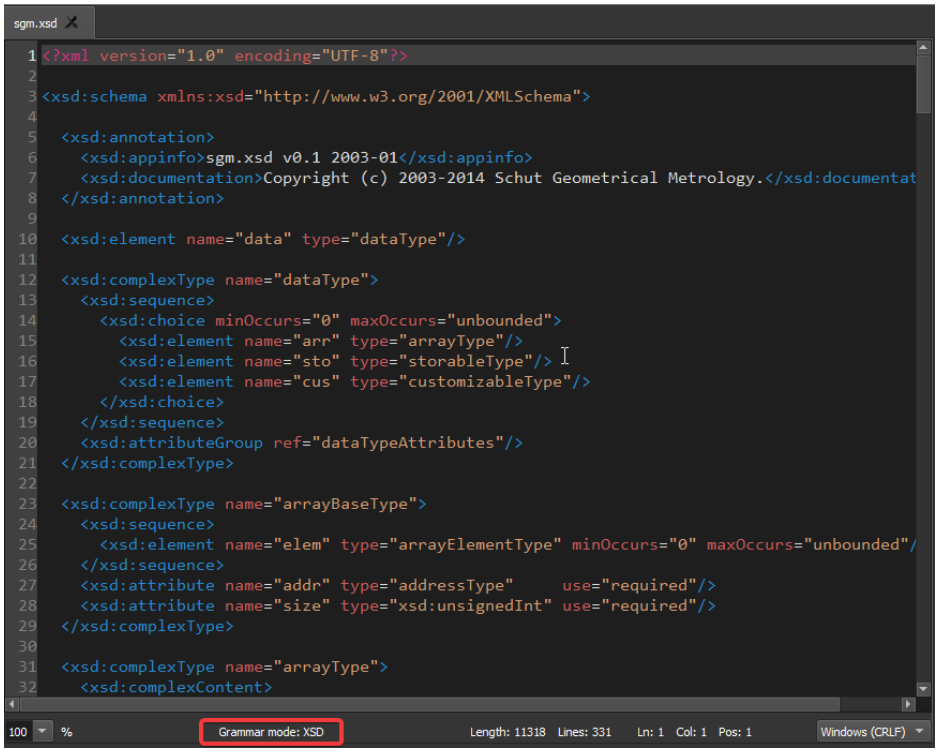

Figure 19: The parser builder in XSD mode

As can be seen, the XSD file is also highlighted differently compared to UBNF. The parser builder essentially changes two things when it changes to a different mode:

1. The parser builder pipeline (What code to execute to turn the file into a parser).
2. The syntax highlighter being used by the text editor.

For XSD, a syntax highlighter was written using regular expressions. It essentially consists of two parts. The first part matches regular expressions and highlights sections of the document based on the regular expression being matched, and then on top is a list of keywords that override this and will always be coloured red (for instance: name and type in the example above).

Regular expression	Colour
$</?[\w:]+>?$	Blue
$/?>$	Blue
$\"[^\"]*(?:[^\\]\|\\\\)\">$	Orange
$\?>$	Magenta
$<\?xml$	Magenta

Figure 20: Regular expressions used for syntax highlighting

The following diagram then explains the full workflow of the automated pipeline:

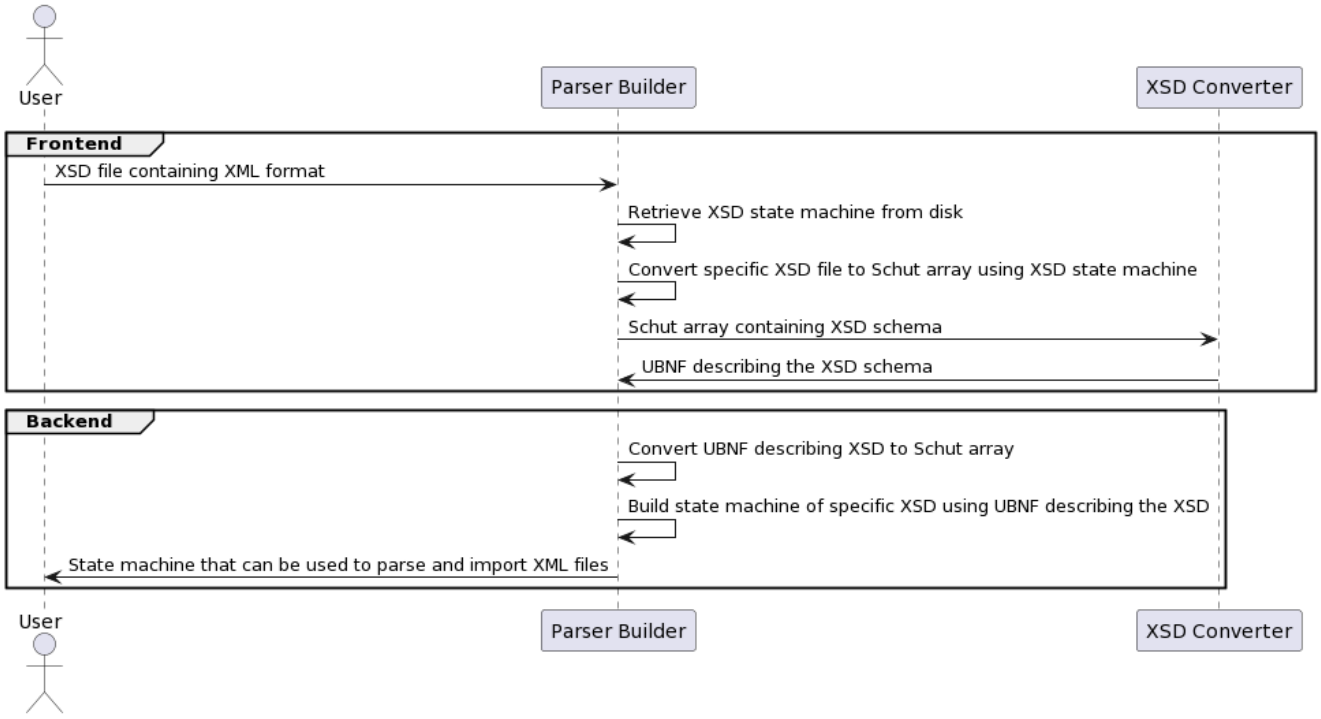

Figure 21: The automated workflow.

4.4 Technology stack

All systems developed during this project are written in C++17. C++20 was not used because the company's infrastructure did not support it yet, and it would have been used if it was a possibility, as many systems within this project would have benefited from C++20's features. While the first stage runs a parser in C++ to parse the XSD, this was already developed by the company and as such this stage was purely written as a UBNF grammar, which is then made into a parser.

The following technologies were used within the project:

Technology	Comment
SmartPointer	Internal technology within Schut. The technology is the company's implementation of the C++ smart pointer feature, and is preferred by the company over C++ smart pointers. While syntactically different, its functionality is the exact same as C++ smart pointers, and is simply used because the company requested so.
Streams	Internal technology within Schut. The technology is the company's implementation of the C++ streams feature, and is preferred by the company over C++ streams. It is syntactically the same as C++ streams, and therefore offers no interesting difference. It is purely used because of the company's request to do so.
GenericParser	Internal technology within Schut. This is the technology that actually runs the parser. It requires the two state machines to be defined, and then uses those state machines to parse a file. Originally developed by Schut, but modified and improved by me.
Array	Internal technology within Schut. This technology defines the Schut Arrays. It provides the data structure, and offers tool to interact with the data structure, including more advanced tools like recursive iterators.

Figure 22: The state machines generated using the grammar from appendix A

5 Parser building

5.1 Summary

This section describes the third and final stage of the pipeline, using the UBNF grammar to generate the final product. Development of the parser generator starts with the back-end, as the back-end is multi-functional in nature, in the sense that it is both part of the pipeline of the parser generator, and it can be used to generate part of the front-end. Schut already has a suitable, but experimental, back-end in place, allowing one to use UBNF to generate the state machines necessary to run a parser. As such, the focus of back-end development is analysing and understanding the existing system, finding flaws in the system, and improving upon them.

One will find a full analysis of the limitations of the toolset in section 8.2. This section will then concern itself with the workings of the parser builder, and the architecture of the improvements made to the tool to resolve such limitations. It may be useful to read the analysis of the tool's limitations prior to reading the improvements made. This section also describes the working of the state machines generated by the tool, and how such state machines are generated.

5.2 State machine

The core of a parser is the state machine, which is explained in section 2.5. A parser essentially just follows the transitions of the state machine to process a file. For the parsers generated in this project, there are two state machines that together form the full final product: The terminal state machine and the grammar state machine.

The following diagram shows the architectural working of a parser generated by the parser builder:

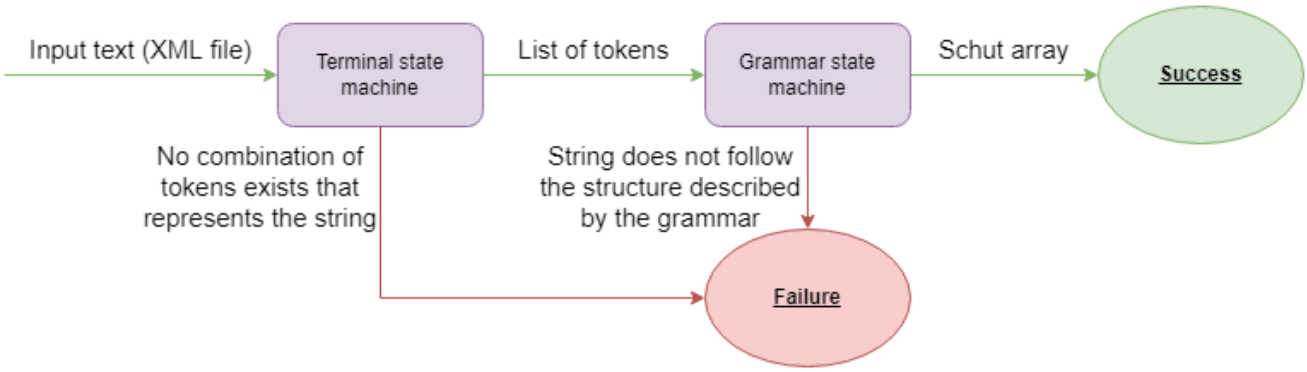

Figure 23: The architecture of a generated parser.

The text is first split up into individual tokens. These tokens are then grouped as a large sequential list, describing the entire text. The parser can then go over the tokens sequentially, matching them against the grammar specifying the text structure. The following figures show an example grammar, and a file following the specification:

```
<$string> := {%Letter}+;

<Circus> :=
"<Circus>"
<Tent> <Stage> <Stand>
"</Circus>";

<Tent> := "<Tent>" <$string> "</tent>";
<Stage> := "<Stage>" <$string> "</Stage>";
<Stand> := "<Stand>" <$string> "</Stand>";
```

Figure 24: Grammar of an example XML file describing a very basic circus.

```
<Circus>
  <Tent>
    ARedTent
  </Tent>
  <Stage>
    AWoodenStage
  <Stand>
    AMetalStand
  </Stand>
</Circus>
```

Figure 25: Implementation of an example XML file describing a very basic circus.

Below is an explanation of the steps taken by the parser to parse this file using the grammar. Note that the reader is not yet expected to understand how it performs these steps, as this is explained in section 5.2.1 and section 5.2.2, however it is important to be able to follow the steps it is taken:

Step 1: Convert string to list of tokens using terminal state machine
Token sequence:

[<Circus>, <Tent>, String (ARedTent), </Tent>, <Stage>, String (AWoodenStage), <Stand>, String (AMetalStand), </Stand>, </Circus>]

Note that the words matched by the "$string" terminal were marked as being a string, where the parser simply keeps track of the value so it can store it into the output Schut array later.

Step 2: Parse list of tokens using the grammar state machine

```
1. Circus {                                                    (Open circus grammar rule)
2. Circus { <Circus>                                           (Read first token)
3. Circus { <Circus> Tent {                                    (Open tent grammar rule)
4. Circus { <Circus> Tent { <Tent>                             (Read second token)
5. Circus { <Circus> Tent { <Tent> String                      (Read third token)
6. Circus { <Circus> Tent { <Tent> String </Tent> }            (Read fourth token)
7. Circus { <Circus> Tent                                      (Tent grammar rule matched)
8. Circus { <Circus> Tent Stage {                              (Open stage grammar rule)
9. Circus { <Circus> Tent Stage { <Stage>                      (Read fifth token)
10. Circus { <Circus> Tent Stage { <Stage> String              (Read sixth token)
11. Circus { <Circus> Tent Stage { <Stage> String </Stage }    (Read seventh token)
12. Circus { <Circus> Tent Stage                               (Stage grammar rule matched)
13. Circus { <Circus> Tent Stage Stand {                       (Open stand grammar rule)
14. Circus { <Circus> Tent Stage Stand { <Stand>               (Read eigth token)
15. Circus { <Circus> Tent Stage Stand { <Stand> String        (Read ninth token)
16. Circus { <Circus> Tent Stage Stand { <Stand String </Stand> }  (Read tenth token)
17. Circus { <Circus> Tent Stage Stand                         (Stand grammar rule matched)
18. Circus { <Circus> Tent Stage Stand </Circus> }             (Read eleventh token)
19. Circus                                                     (Circus grammar rule matched)
```

As Circus is the root element of this grammar, the file is accepted as Circus is the only remaining rule at the end of

the parsing process. The large question that now remains is how it performs these 2 steps using the state machines.

5.2.1 Terminal state machine

The terminal state machine is used to generate the tokens from the input string. In order to understand how one works, it is useful to see an example. The following state machine is the terminal state machine of the Circus example used throughout this section:

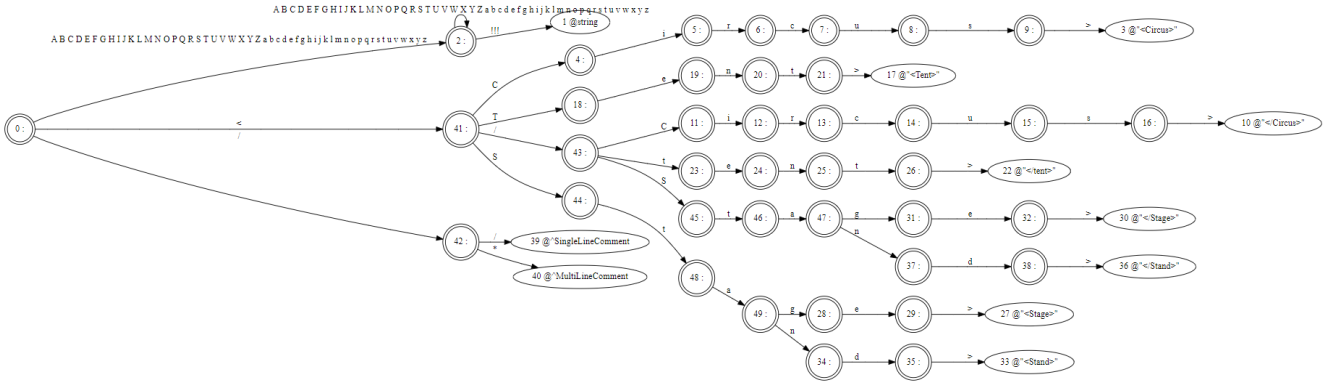

Figure 26: The terminal state machine for the circus grammar.

The terminal state machine's root state is state 0, which is all the way on the left side. The process starts in state 0. From state 0, one can travel to the other states by reading in a character from the string. The machine then follows the transition to another state that contains the next character read in from the string. Eventually, one will reach a terminal state, which are the states containing a "@" symbol. These states represent actual valid tokens. The tokens read in so far are removed from the string, and the token displayed in the terminal state is returned. The next token can then be generated by resetting the machine to state 0, and starting the process all over again. One can do this until no more characters remain in the input string.

The state machine will always match the longest possible string, which means that if the state machine can choose between multiple tokens, it will always take the longest possible one. An example within this state machine is the "@string" token in state 1. It can at any point from state 2 transition to state 1 and terminate, but it will keep reading in characters of the alphabet until it can no longer, and then transition to state 1.

5.2.2 Grammar state machine

Where the terminal state machine can be used to generate the tokens, the grammar state machine represents the grammar used to define the format of the file. It can, just like the terminal state machines, be transitioned over, but unlike the terminal state machine, it will only run once, and will use tokens rather than characters to transition to new states. The following image shows the grammar state machine of the circus example:

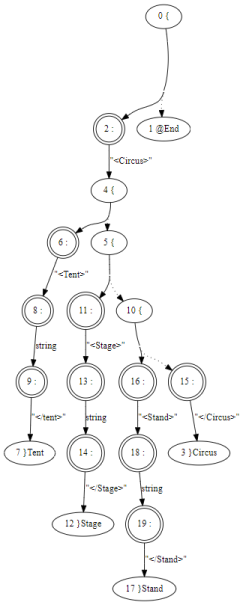

Figure 27: The grammar state machine for the circus grammar.

Yet again, state 0 is our root state. From this state, two types of transitions appear: a continuous line and a dotted line. The process of traversing the grammar state machine is a recursive one, which means it will travel back when a grammar rule has been matched. The continuous line describes the direction the state machine will travel immediately, while the dotted line describes the direction that the state machine will travel once it has returned to the state. So from state 0, it will first travel to state 2, and once this state has accomplished its task, it will return to state 0, at which point it will travel the dotted line to state 1, which is the @End rule, meaning that the parsing is complete.

So the state machine will first travel to state 2, starting the Circus rule, it will then start matching tokens, travelling to state 7, at which point it has finished the Tent rule, it then recurses to state 4, at which point it will take the new dotted route, state 5. If one were to continue this process, one would see that it first opens the circus rule. It then continued by opening the tent rule, followed by the stage rule, and finally followed by the stand rule. It then finished by closing the circus rule, completing the grammar.

5.3 Improvements

This section will describe some of the improvements made to the company's experimental toolset based on the analysis performed in section 8.

5.3.1 Error handling

The first project to improve the usability of the parser builder was to rework parts of the error handler. This project was a direct response to issues discovered by our analysis described in section 8.2.2.1. While a more permanent solution was produced by the work described in section 5.3.2, this project was performed first as the tokenisation project was complex to the point that it wasn't clear whether it would be possible, and in general the error messages were lacking in detail.

The new error messages look as follows:

```
Parsing error (25,19): Expected: ~":choice" or ~":sequence>" but I read: ~":element"
      <xsd:element name="elem" type="arrayElementType" minOccurs="0" maxOccurs="unbounded"/>
                   ^
Parsing text failed.
```

Figure 28: The error message produced, showing the programmer exactly what was going wrong.

Especially interesting here is the "Expected, but I read" section of the error message. Prior to the new error messages, it was really hard to track why the grammar didn't accept a certain token, because sometimes the grammar inputted makes sense for a human, but then due to the technicalities of the parser builder, it would break. Here it however exactly shows which tokens it would have accepted, and which token it instead read. If one defined a token on the other end of the file that was accidentally getting matched here, it would be immediately obvious.

5.3.2 Tokenisation

One of the most critical flaws found during our analysis of the company's tools belongs to the tokeniser of the state machines generated by the parser builder. This analysis can be located in section 8.2.2.1, and as such this section will purely concern itself with the implemented solution.

The solution is to limit the scope of tokens considered during the process. Essentially, the tokeniser should pick the longest token available that the parser can actually match within its current production rule, rather than the longest globally available token.

How to actually implement this becomes clear when one were to look at an example state diagram where one would need to take precautions to ensure the grammar behaves properly:

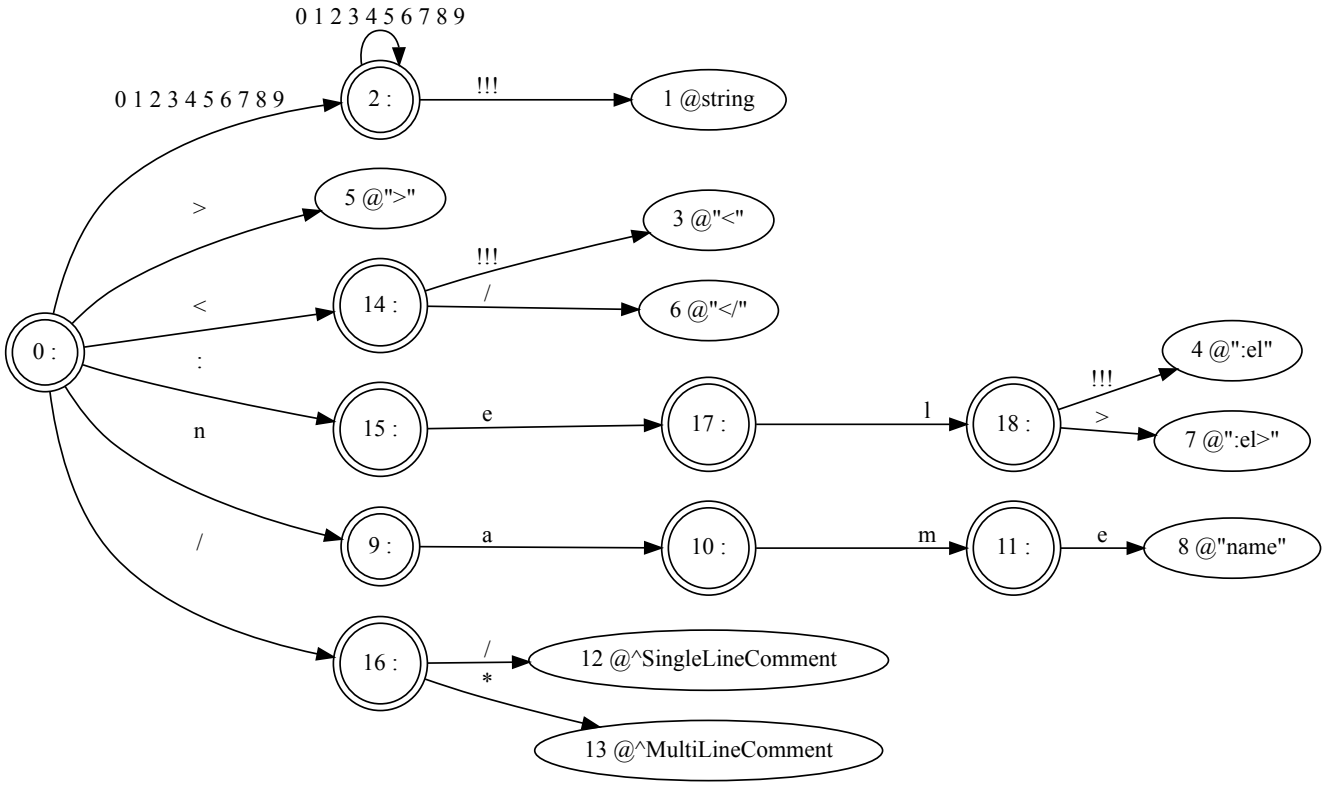

Figure 29: Example terminal state diagram that would misbehave with a global tokeniser.

The interesting transitions are the "!!!" transitions. These transitions state that one has a valid token and can stop early here if they want to, though they can also keep going to match a longer token. If we were to have a situation where the tokeniser picks a longer token that the parser cannot match, then the shorter token that can be matched surely ends in a "!!!" transition, as there otherwise would not be a longer token. As such, the algorithm should keep track of any "!!!" transitions it encounters along the way, check if the token thus far is a token that is useful to the parser, and keep track of the longest useful one. Once it intends to return the globally longest token, it checks whether it is useful to the parser, and if it is not, it takes the longest useful token if it exists. If it does not exist, it will simply return the longest global token, and the parser will fail.

One has to take care to revert the tokeniser's state to the state it was in when it read the longest useful token. This means that all references to the current token need to be updated, the marker of how far the line has been read has to be moved back, and the file stream from which the characters are being read has to have their unused tokens be inserted back into the front.

There is however a catch with more complicated terminal state machines. For example, let us investigate the following state machine:

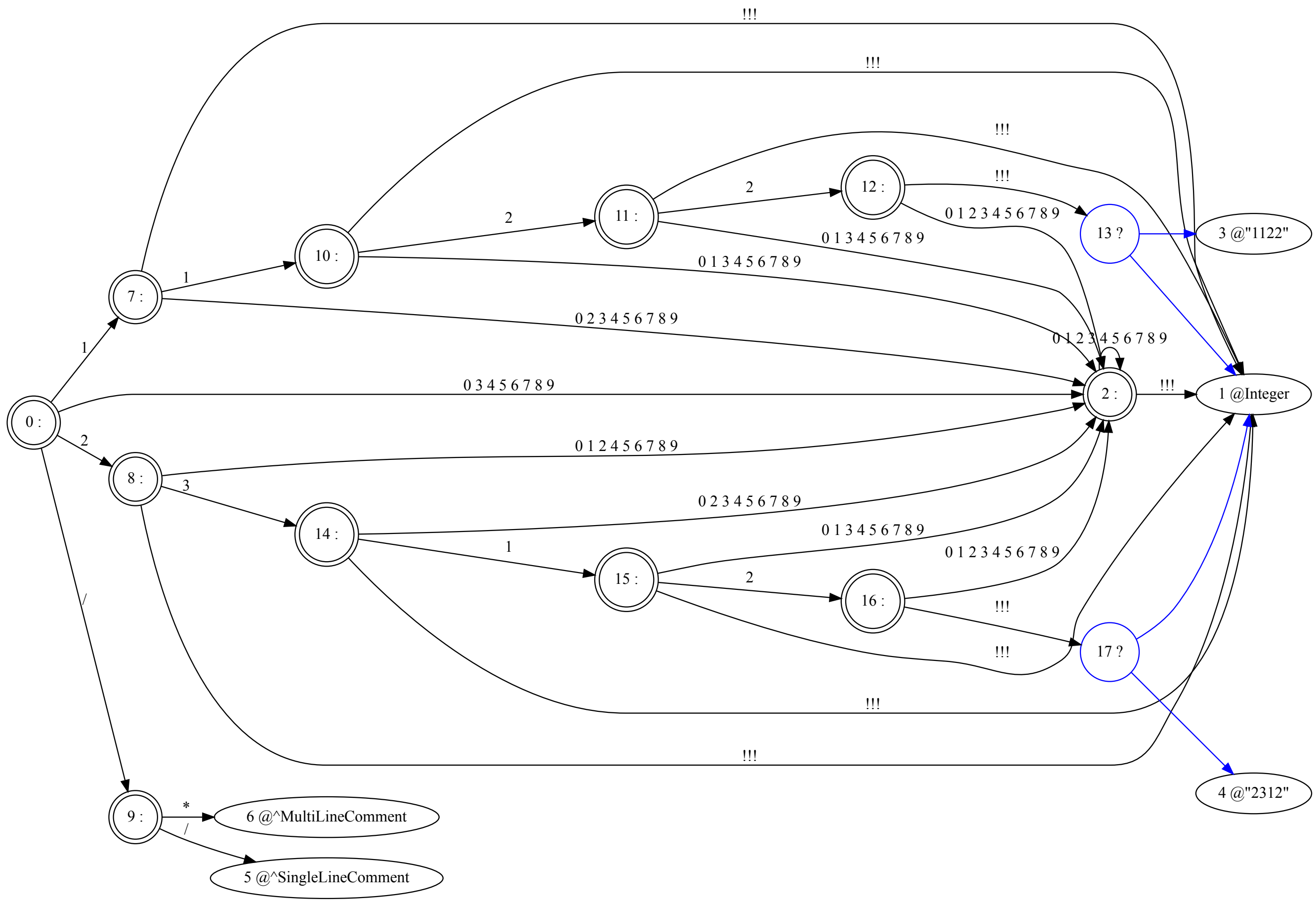

Figure 30: Example terminal state machine accepting 3 tokens: ”2312”, ”1122” and any integer.

This relatively simple state machine, has an interesting property, the blue transitions. These represent OR transitions, where the traveller is allowed to go in both directions. These are logical, because the token ”1122” is also accepted as an integer So if we read 1122, we can match either the 1122 token, or the integer. In order to detect any usable tokens, it is important to ensure that the algorithm recursively explores all OR transitions when it attempts to determine the usable tokens accepted through the current string.

5.3.3 Encodings

Another flaw detected within the parser builder detected during analysis was the fact that any text it imported from a file was assumed to be UTF-8. This was described in section 8.2.1.1. This was resolved by detected the encoding of the text, and then select the right decoder based on the encoding. Normally this can be handled by investigating the first few bytes of a file, as this tends to specify the encoding being used. It is however not guaranteed that these marker bytes are present, so it is sometimes hard to determine the type. Schut however already developed a library that can be used to automatically detect the encoding used within the tool, which attempts to determine the encoding based on the first few bytes, and tries to make a best guess if these bytes do not exist. As such, fixing this issue was simply a matter of verifying that Schut's solution was correct and was what I needed, and then using their library to detect the encoding, and make a decision on the right decoder based on this.

5.3.4 Static Content

Section 8.2.3.3 notes that another issue exhibited by the parser builder: The lack of the ability to add static data to the output Schut array. In order to resolve this, I added a feature to the UBNF syntax in order to be able to support static data. Below is an example of these features:

```
<Root> := <a> <b>;
<a; StaticContent="First"; StaticContentName="Order"> := "a";
<b; StaticContent="Second"; StaticContentName="Order"> := "b";
```

Figure 31: An example of the static content feature.

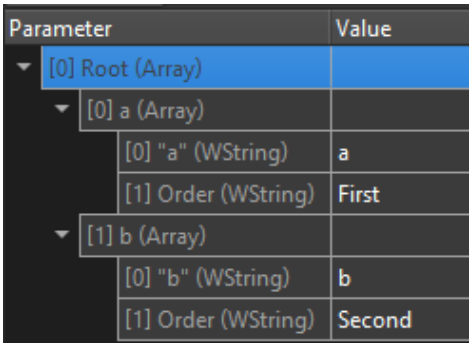

Figure 32: The output of the parser with the input string "ab".

A user of the grammar can use the "StaticContent" parameter of a rule to add a string to the Schut Array entry representing that rule, while the user can use the "StaticContentName" parameter to give a name to this added data. This means that a user can only add 1 string to the Schut Array, which is not ideal as it would be more ideal if the user could add multiple strings, but the current implementation of UBNF makes that exceedingly difficult.

5.3.5 Grammar parameters

Section 8.2.2.4 stated that the system was using an unnecessary amount of data by storing all parameters for every grammar production rule, no matter if they actually are used or not. I resolved this by implementing a system that only adds parameters when they are necessary. There is however a catch: Boolean parameters are used directly in logic throughout the program, and are not checked for their existence prior to usage. Checking for existance of these parameters would actually increase execution time for the program, and as such may actually result in a performance degredation. As such, only the non-boolean parameters are selectively added based on whether they are defined. The following shows the difference in data size of the end of rule node between the old version and the new version on a node that has no parameters defined:

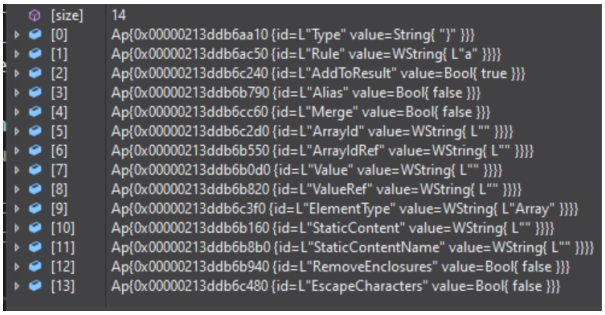

Figure 33: The old system for storing parameters.

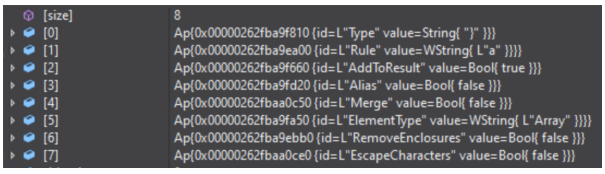

Figure 34: The new system for storing parameters.

5.3.6 Parameter extension

9 new parameters were added to the UBNF grammar. The following table describes these parameters together with their purpose:

Parameter	Incompatible with	Purpose
StaticContent	None	Allows one to add data to the data structure without it actually appearing in the text that is being parsed. Useful to for example define the datatype of another entry.
StaticContentName	None	Assign a name to the data entry added by StaticContent
MaxInclusive	MaxExclusive	Specifiy a maximum bound for a numeric value in the result data structure, the bound itself is included.
MaxExclusive	MaxInclusive	Specify a maximum bound for a numeric value in the result data structure, the bound itself is excluded.
MinInclusive	MinExclusive	Specifiy a minimum bound for a numeric value in the result data structure, the bound itself is included.
MinExclusive	MinInclusive	Specify a minimum bound for a numeric value in the result data structure, the bound itself is excluded.
Length	MinLength & MaxLength	Specify an exact expected length of a string in the result data structure.
MinLength	Length	Specify a maximum expected length of a string in the result data structure.
MaxLength	Length	Specify a minimum expected length of a string in the result data structure.

Figure 35: The state machines generated using the grammar from appendix A

UBNF is defined using UBNF. As such, adding these parameters was first a matter of updating the UBNF grammar to actually allow for these new keywords. One very complex feature to support here were the bounds checks for numeric values. There are two types: Floating point numeric values and integer numeric values. Both have different syntax, where floating point bounds may, for example, use floating point numbers and scientific notation, while integer numeric values may not. Furthermore, they also have to be stored in different data types, as floating point numeric values have to be stored as doubles, while integer numeric values have to be stored as longs. However, we do not want to add a separate parameter for each type, and therefore we want the grammar to be able to recognise this automatically.

Once this was defined, the logic had to be implemented in the source code of the parser builder. The main things to add were extensions to the system that retrieves the parameters from the input and stores them in their associated rules in the generated parsers and adding the logic for actually checking the bounds, length, and appending static content during parsing of the generated parsers. Both were successfully added; therefore, these nine parameters work without issues.

6 XSD parsing

6.1 Summary

This section describes the first stage of the pipeline, parsing the XSD file and converting it into a machine-readable format. This parser is generated using the back-end of the parser generator, which can generate parsers using UBNF to generate a parser that parses XSD. This approach can be achieved by assembling a UBNF grammar specification of XSD and using it to generate the state machines for an XSD parser. As the back-end generates a parser that processes the data into a Schut array, we can use the Schut array as our machine-readable format.

By controlling how the data is converted to the Schut array, like removing unnecessary information found during parsing or renaming some data entries, we can create a highly readable format that requires little preparation in the second stage before starting the conversion process. As such, no post-processing would be required by this stage. That said, pre-processing will be required. Firstly, to apply a few transformations to the data to make it easier to parse, and secondly, to deal with includes and imports. An XSD file is allowed to include or import other XSD files. All files must be appended together before producing the final Schut array.

6.2 Introduction

There are multiple ways to design a parser that can read an XSD file. XSD is an XML format, so one of the most obvious solutions is to use a general-purpose XML parser to read in an XSD file and then match the tags. This approach does have a few drawbacks:

- XML general-purpose parsers are notoriously slow [12].
- The structure of the XSD will be hard to trace back due to it being embedded in code interacting with the XML parser.
- It offers little control over the XSD structure. Many tags within XSD are irrelevant and should be thrown away prior to converting the structure to UBNF.

We will be using the back-end of the parser generator to generate the XSD parser used within that same parser generator. So while the back-end is meant to receive auto-generated UBNF definitions of XML files, we will manually craft a definition of XSD in UBNF and then pass this to the back-end. Performing these steps will result in a parser that can parse XSD and import the data into a Schut array, which can then be machine-read.

UBNF allows many details to be specified within this grammar. We can use these details to modify the Schut array produced by the generated parser to ignore certain elements of the XSD, rename elements, or change the structure. These specified details allow us to create a grammar that reads in an XSD and optimises its structure to make it more readable to the tool converting the Schut array containing the XSD to UBNF.

Therefore, our goal is to write a UBNF grammar that matches the XSD format and performs optimisations on the format. This grammar is provided in Appendix A of this document.

6.3 XSD hierarchy

In order to be able to parse the XSD format fully, it is critical to understand the structure of the format and which elements can inherit from which other elements. This section will show the XSD format's structure and the elements' hierarchical tree. This section will concern itself purely with inheritance, which is the core of building an XSD parser.

The actual inheritance graph of XSD would get enormously large. As such, the inheritance of individual components is specified, and one can complete the graph by substituting the graphs.

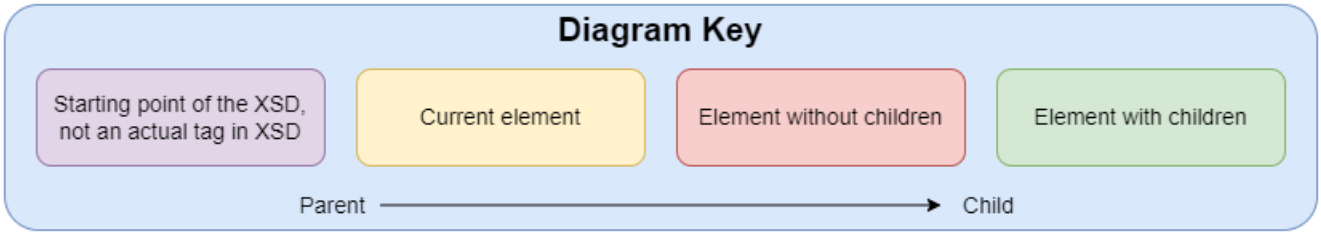

Figure 36: The key for the diagrams explaining the tree structure.

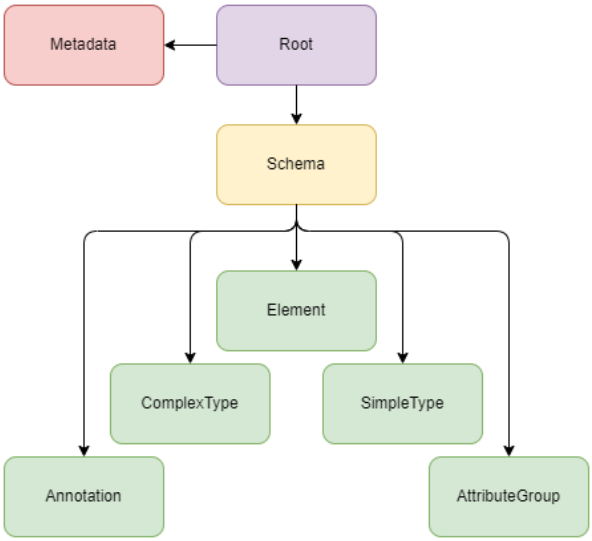

Figure 37: The root of an XSD file structure.

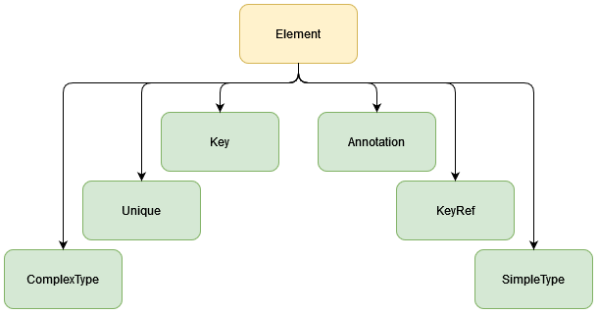

Figure 38: The element tag within XSD.

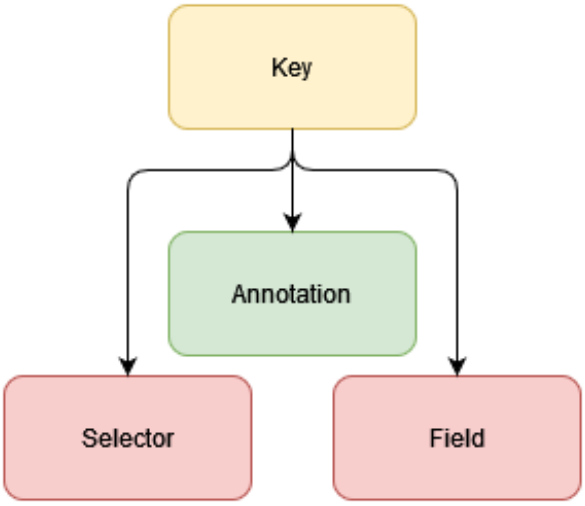

Figure 39: The key tag within XSD.

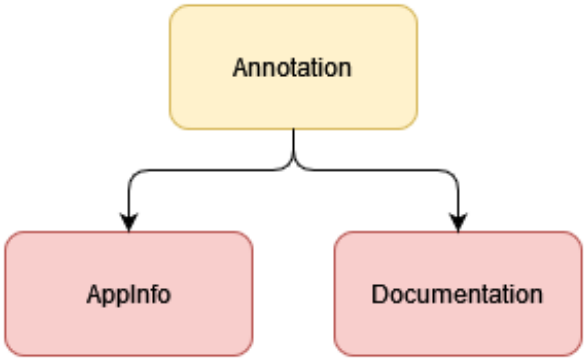

Figure 40: The annotation tag within XSD.

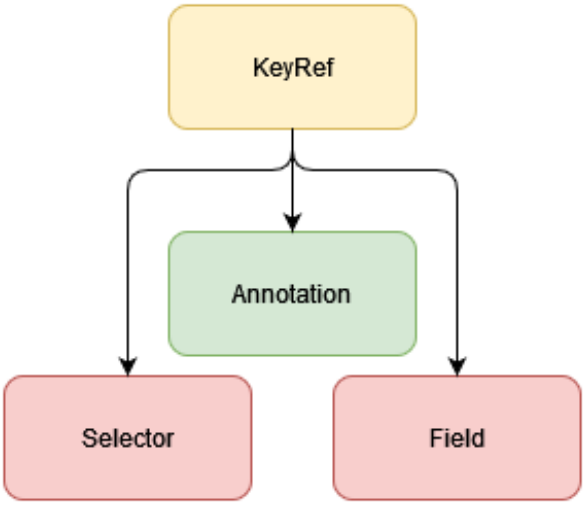

Figure 41: The keyRef tag within XSD.

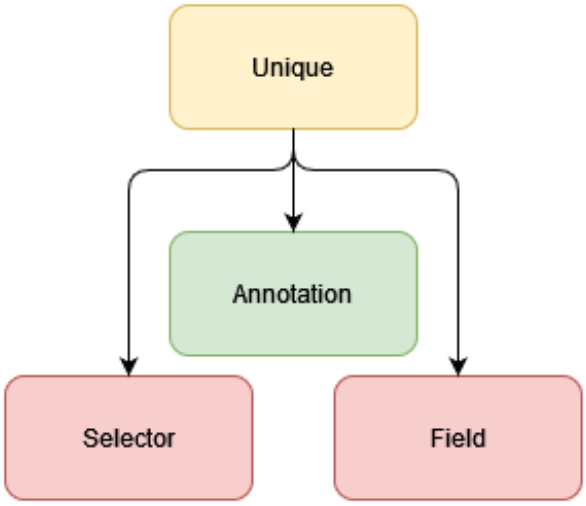

Figure 42: The unique tag within XSD.

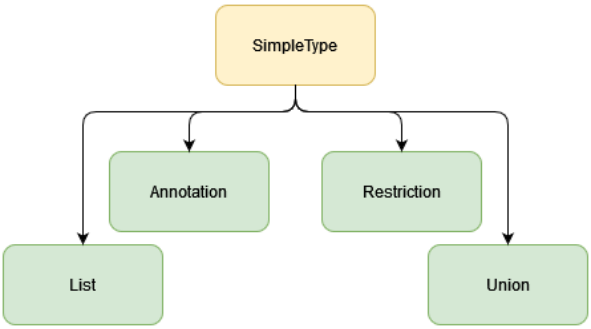

Figure 43: The simpleType tag within XSD.

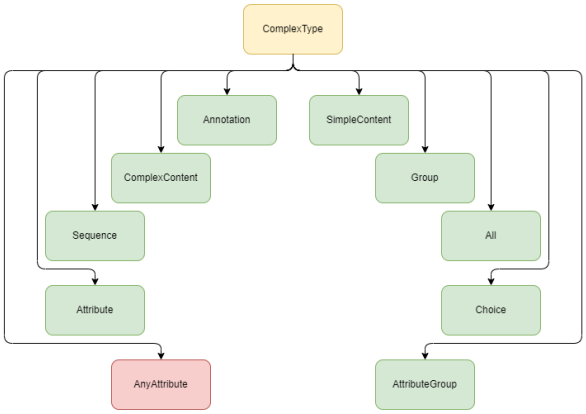

Figure 44: The complexType tag within XSD.

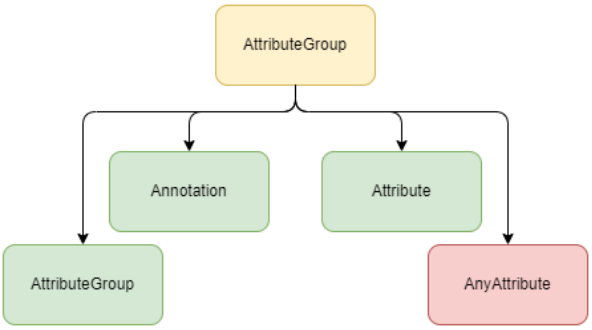

Figure 45: The attributeGroup tag within XSD.

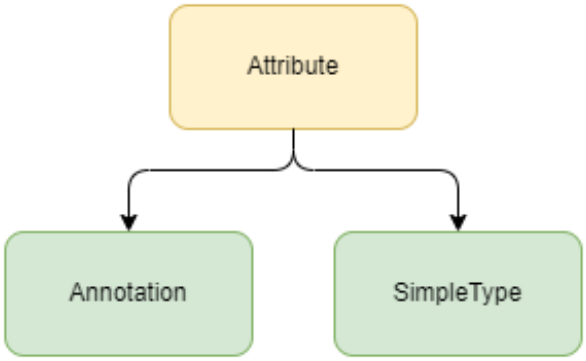

Figure 46: The attribute tag within XSD.

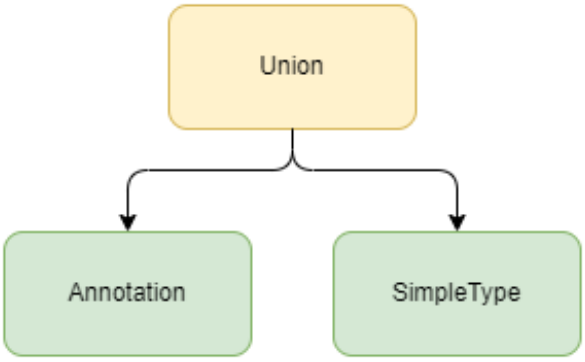

Figure 47: The union tag within XSD.

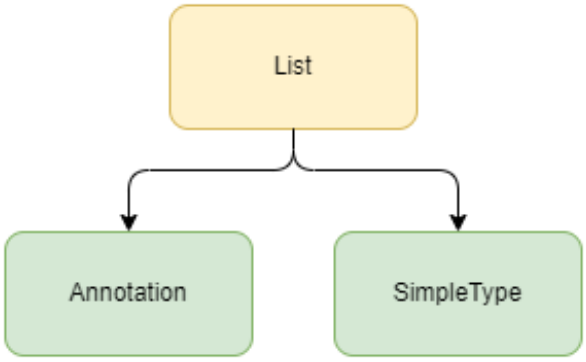

Figure 48: The list tag within XSD.

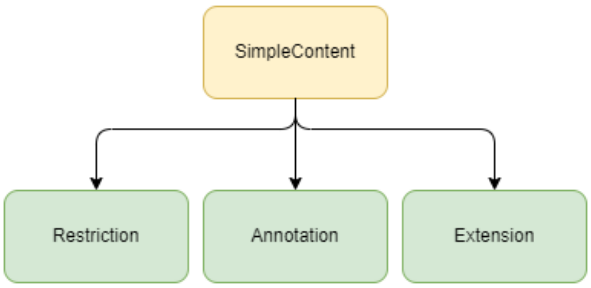

Figure 49: The simpleContent tag within XSD.

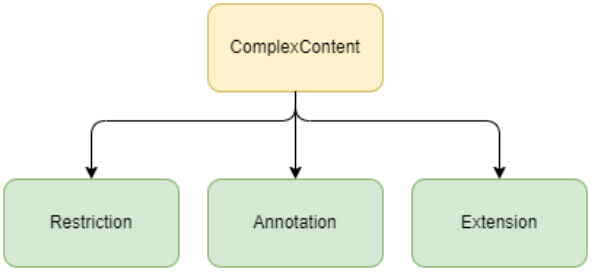

Figure 50: The complexContent tag within XSD.

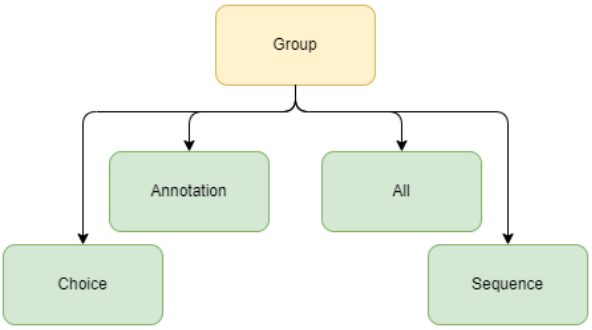

Figure 51: The group tag within XSD.

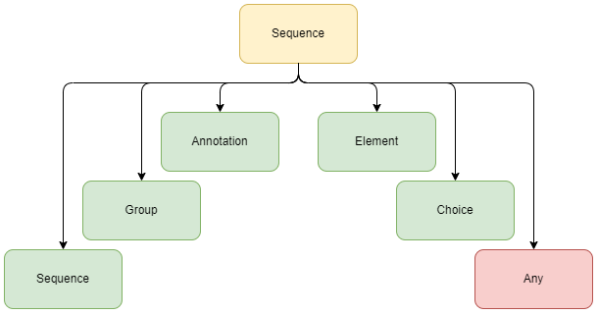

Figure 52: The sequence tag within XSD.

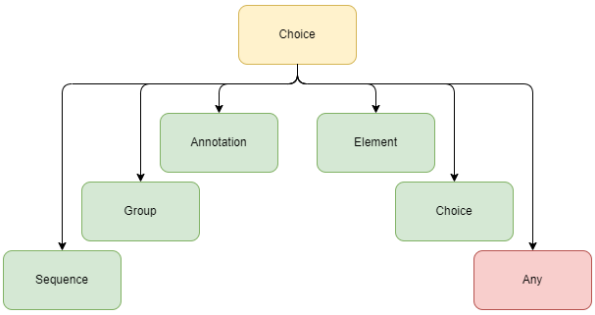

Figure 53: The choice tag within XSD.

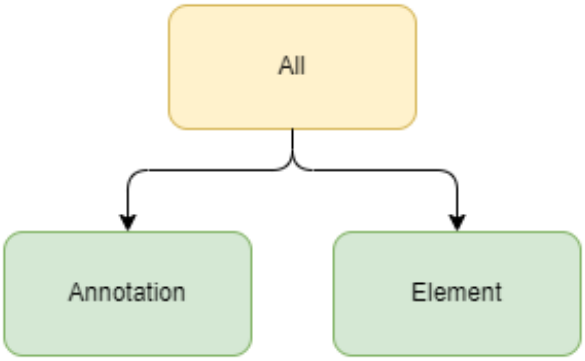

Figure 54: The all tag within XSD.

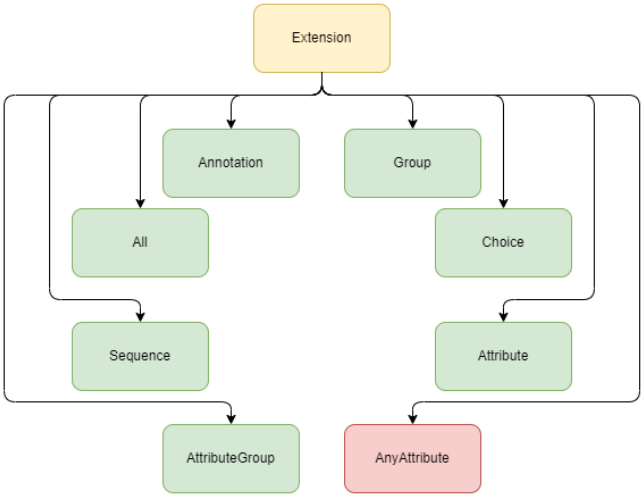

Figure 55: The extension tag within XSD.

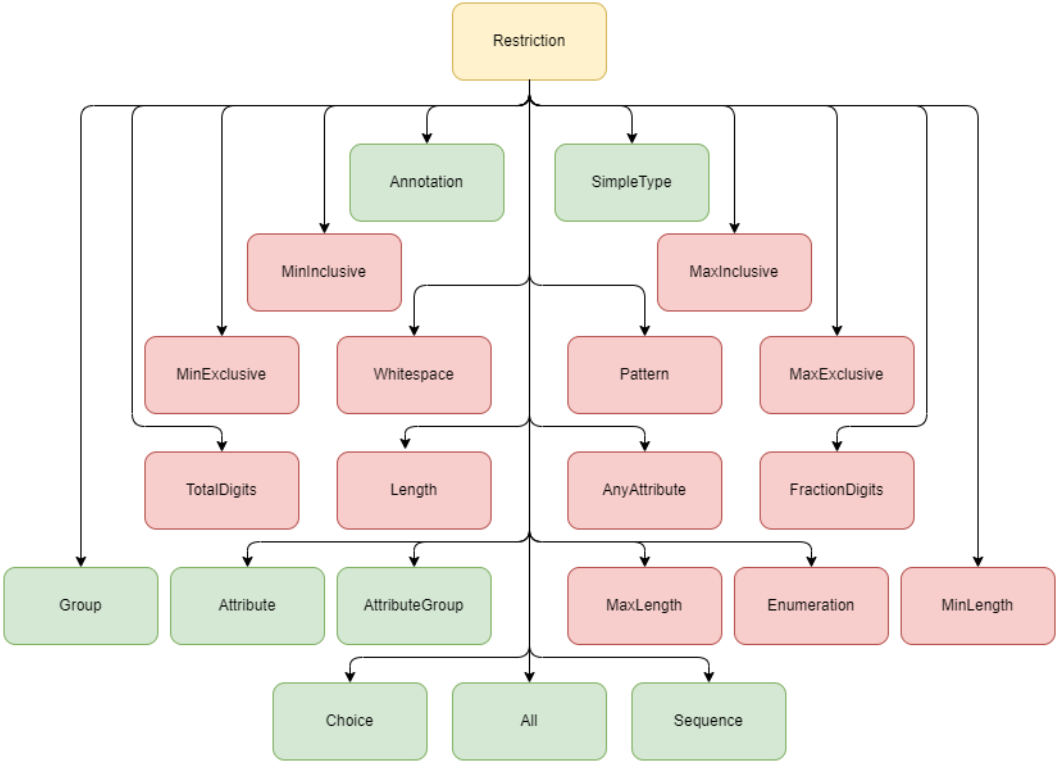

Figure 56: The restriction tag within XSD.

6.4 Pre-processor

XSD files require pre-processing before being parsed. This pre-processing is necessary because a few operations have to be applied to the files that can only be done through writing code for it, rather than specifying the changes within the grammar, as code is more expressive than the grammar.

The first issue to resolve is the issue of annotations. Annotations provide documentation about the element which contains the annotation. Its fundamental issue is that an annotation can contain any character as its content. As a result, writing a terminal state that describes the content of the annotation would match the entire file as one enormous token. This issue can be resolved by specifying that the token has to start with the annotation opening tag and end with the annotation closing tag. This solution, however, has its own issues.

The issue that results from this approach is that it will match the first annotation opening tag in the file with the last annotation closing tag, which would very likely see a considerable part of the XSD file as an annotation, as seen in the following example:

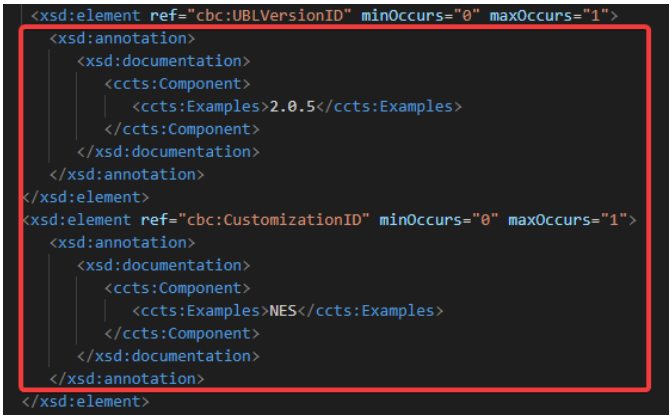

Figure 57: The red area would be matched as a single annotation.

As such, without the availability of a non-greedy operator described in section 8.2.3.2 we need to result to pre-prossing to manually remove all annotations from the file prior to parsing the XSD.

6.5 Post-processor

Another larger task that must be performed outside of the state machine is the task of handling imports. Files containing XSD, just like many of the files containing programming languages, can import other files. This way, a large project can be split into multiple files that can import each other. The following snippet of an XSD file shows what that looks like:

```
<xsd:import namespace="urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2"
            schemaLocation="../common/UBL-CommonAggregateComponents-2.1.xsd"/>
<xsd:import namespace="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2"
            schemaLocation="../common/UBL-CommonBasicComponents-2.1.xsd"/>
<xsd:import namespace="urn:oasis:names:specification:ubl:schema:xsd:CommonExtensionComponents-2"
            schemaLocation="../common/UBL-CommonExtensionComponents-2.1.xsd"/>
```

Figure 58: A snippet from a large XSD project containing imports.

The schemaLocation attribute contains the relative path to the file being imported.

The following diagram shows how these imports are handled:

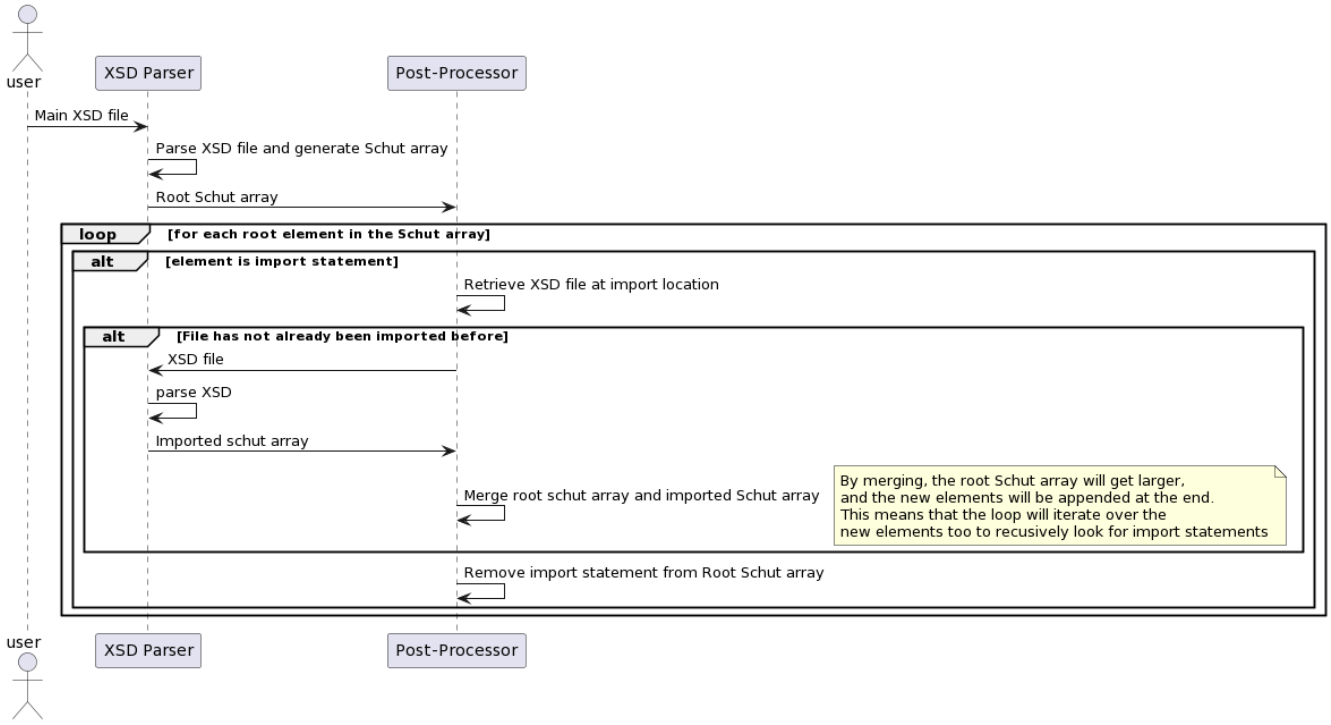

Figure 59: The sequence diagram of the postprocessing after parsing.

6.6 XSD defined in UBNF

Knowing the hierarchical structure of the XSD grammar and the attributes contained within the tags, the grammar describing the XSD can be written. Appendix A of this document contains the complete grammar describing the XSD. It consists of 3 sections: The metadata, the terminals, and the non-terminals.

The metadata is a section that contains a few parameters that can be set for the UBNF parser, like how comments are defined so that the parser can ignore them. It also defines the start rule, essentially the root of the file, from which the grammar rules are expanded to parse the file.

The terminals describe tokens contained within the file. They only define more complex tokens that require regular expressions, as tokens defined as simple static strings can be defined in-line in UBNF. It consists of two sections. The first is the character sets, enclosed by "{}". Such statements describe a set of characters, which the actual tokens can then use to describe words. Then the actual terminals appear, which are enclosed by "<$" and ">". They define actual tokens which are to be used by the grammar rules.

The non-terminals form the core of the grammar. The rules define how the file is structured and which components may follow other components. As explained in the UBNF background, note that many rules contain parameters that control the output Schut array. These parameters allow us to rename, remove or replace items to

get a logical structure that is nice to parse.

One issue with creating a grammar that parses the entire XSD format is that many edge cases may be hard to discover. While all children contained within a section were worked out in section 6.3, there are further details that one would need to deal with, like whether a child is optional or mandatory, how many times it is allowed to appear, and the order of the children or lack thereof. The official specification for XSD contains exact regular expressions for the children of every XSD element. Given that UBNF supports regular expressions within the grammar, it was easy to copy this over to the grammar used within this project, guaranteeing that every case is covered.

6.7 State machines

The UBNF grammar itself, however, does not represent much detail about the architectural implementation of the parser. However, this detail is represented by the state machines. The following links provide access to the graph representation of the state machines of the XSD parser, which are included externally because of their immense size:

State machine	Link
Non-terminal state machine	https://github.com/LarsSven/Thesis/blob/master/Machines/Terminal_Machine.svg
Terminal state machine	https://github.com/LarsSven/Thesis/blob/master/Machines/Grammar_Machine.svg

Figure 60: The state machines generated using the grammar from appendix A

6.8 Output

In order to see what the parser builder does with this UBNF, an example of a simple XSD file and the resulting Schut array will be shown. The following XSD is the input to the parser generated by the parser builder using the UBNF definition of XSD:

```
<?xml version="1.0" encoding="UTF−8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Figure 61: Example adapted from https://www.w3schools.com/xml/schema_example.asp.

The following Schut array is produced by the output of this first stage:

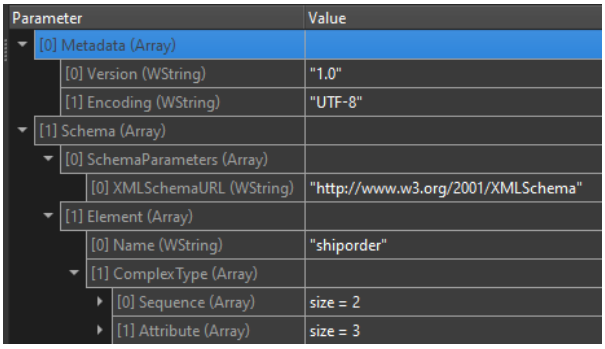

Figure 62: The root structure of the Schut array.

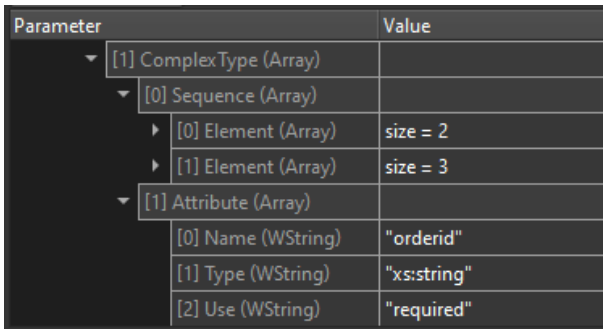

Figure 63: The expanded complex type seen collapsed in the root.

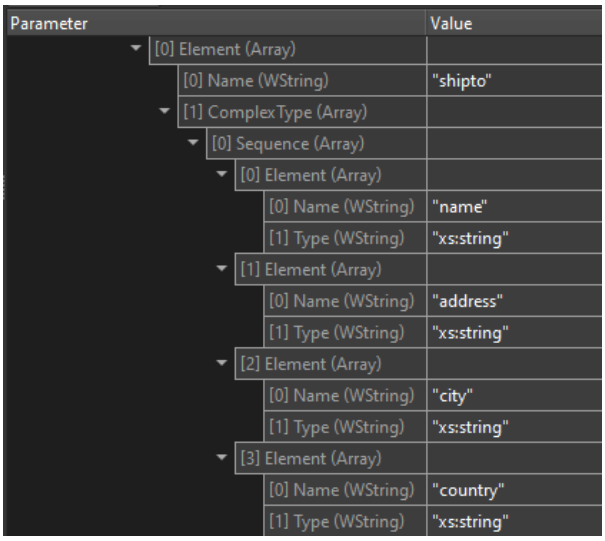

Figure 64: The first element contained within the complex type.

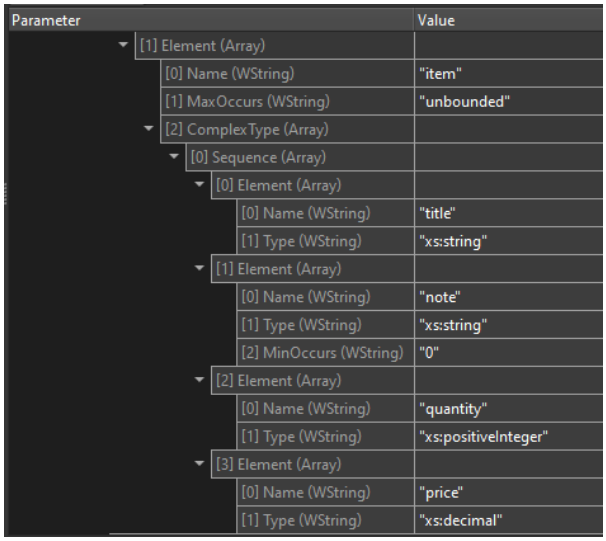

Figure 65: The second element contained within the complex type.

This Schut array clearly provides structured, hierarchical access to the data contained within the file. The program's second stage can then iterate over this, access the internal data, and use this to generate the UBNF grammar defining the same grammar scheme.

7 XSD conversion

7.1 Summary

This section describes the second stage of the pipeline, converting XSD to UBNF. One can provide the converter with a Schut array generated by the XSD parser described in section 6, which the converter will then turn into a UBNF grammar representing this exact same format. This stage will essentially be a C++ library that provides a converter class. This class contains one public function, which is utilised to convert a Schut array to a string containing the UBNF grammar describing it.

The library essentially builds an internal data structure that represents the schema in a format most optimised for writing to UBNF. Essentially, all unnecessary syntax is thrown away, and syntax that is hard to convert to UBNF is first reworked. Once the data structure is fully assembled, the UBNF is written to a file by traversing the data structure. This process is further explained in section 7.2 and section 7.4.

7.2 Conversion process

The conversion process works through a 3-step process. The first step is construction, where the tool uses the Schut array received as input to construct a tree of elements that should be in the final UBNF grammar. Every part of the XSD relevant to the final UBNF grammar got a C++ class designed for it. These classes contain the data necessary to write the UBNF grammar representing it. For example, the XSD element is represented through the "Element" class, which contains the element's name, the datatype of the content, an array of attributes, and an array of children to the element. This data is all that is necessary to build that element.

The second step is post-processing. The reasoning for this step to exist is quite specific to XSD, so it is discussed in section 7.4. However, as a short summary, the data structure cannot be assembled by just sequentially going over the Schut array. Hence, certain choices in construction need to be deferred to the end, when all other elements of the data structure have been assembled, as these choices depend on what is available in the rest of the data structure.

The third step is then writing. The converter now has access to a tree-like data structure of custom classes that each have some direct translation to UBNF. As such, it is subsequently simply a matter of calling the "Write" function contained within each class, which will write the element as UBNF grammar, and recursively call the write on its data members. So for our example with the "Element" class, it would call the write function on all its attributes and children, followed by writing itself to the UBNF target, which is a string containing the grammar in practice.

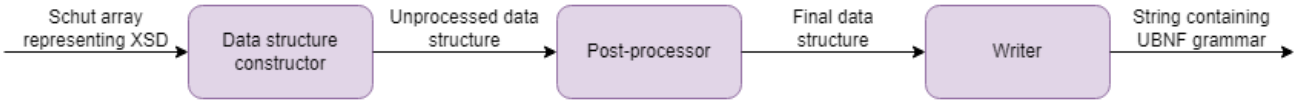

Figure 66: The workings of the conversion process.

The way the data structure is generated is quite simple from a macro perspective and is quite similar for all parts of the XSD. As such, showing an example of one of the data structure building algorithms helps understand the approach taken for all elements. The following snippet is the core processing function of the "restriction" element:

```
void Restriction::process(SpElement target, const sgm::base::Array &restriction) {
    for (size_t idx = 0; idx <restriction.getSize(); ++idx) {
      const std::wstring &xsdIdentifier = restriction.getId(idx);
      // Attribute
      if (xsdIdentifier == Identifiers::BASE) {
        std::pair<std::wstring, std::wstring> type = XsdToUbnfConverter::terminalType(restriction.getWString(idx));
        if (!type.first.empty()) {
          target->setType(type);
        } else if (
            !restriction.isId(Identifiers::SEQUENCE) && !restriction.isId(Identifiers::ALL) &&
            !restriction.isId(Identifiers::CHOICE) && !restriction.isId(Identifiers::GROUP)) {
          XsdToUbnfConverter::addRequest(target.share(), restriction.getWString(idx));
        }
      // Children
      } else if (xsdIdentifier == Identifiers::GROUP) {
        Group::process(*target, restriction.getArray(idx));
      } else if (xsdIdentifier == Identifiers::SEQUENCE) {
        target->addChild(NnspSequence::make(restriction.getArray(idx)));
      } else if (xsdIdentifier == Identifiers::ALL) {
        target->addChild(NnspAll::make(restriction.getArray(idx)));
      } else if (xsdIdentifier == Identifiers::CHOICE) {
        target->addChild(NnspChoice::make(restriction.getArray(idx)));
      } else if (xsdIdentifier == Identifiers::ATTRIBUTE_GROUP) {
        AttributeGroup::process(target.share(), restriction.getArray(idx));
      } else if (xsdIdentifier == Identifiers::ATTRIBUTE) {
        SpAttribute attribute = NnspAttribute::make();
        attribute->construct(restriction.getArray(idx), false);
        target->addChild(std::move(attribute));
      // Constraints
      } else if (xsdIdentifier == Identifiers::MIN_LENGTH) {
        target->addConstraint(Identifiers::MIN_LENGTH, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::MAX_LENGTH) {
        target->addConstraint(Identifiers::MAX_LENGTH, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::MIN_INCLUSIVE) {
        target->addConstraint(Identifiers::MIN_INCLUSIVE, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::MAX_INCLUSIVE) {
        target->addConstraint(Identifiers::MAX_INCLUSIVE, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::MIN_EXCLUSIVE) {
        target->addConstraint(Identifiers::MIN_EXCLUSIVE, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::MAX_EXCLUSIVE) {
        target->addConstraint(Identifiers::MAX_EXCLUSIVE, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::LENGTH) {
        target->addConstraint(Identifiers::LENGTH, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::TOTAL_DIGITS) {
        target->addConstraint(Identifiers::TOTAL_DIGITS, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::FRACTION_DIGITS) {
        target->addConstraint(Identifiers::FRACTION_DIGITS, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::PATTERN) {
        target->addConstraint(Identifiers::PATTERN, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::ENUM) {
        target->addConstraint(Identifiers::ENUM, restriction.getWString(idx));
      } else if (xsdIdentifier == Identifiers::WHTITESPACE) {
        target->addConstraint(Identifiers::WHTITESPACE, restriction.getWString(idx));
      }
    }
  }
```


Figure 67: The processing algorithm for the restriction element.

The restriction itself is not a useful element within our data structure, as it is a part of an element. As such, we immediately collapse the element by simply defining a static "process" function that takes the parent element and then appends the found data to the element information. As such, our two parameters to the function are: A target element to add the data to, and a Schut array which contains the actual data of the array.

The building algorithm contains a relatively simple structure: It loops over the entries within the array using a for loop. We can then retrieve the name of the entry, which is called the identifier. Based on this identifier, we can decide what to do with it, which is handled in the if-else chain. Note that as the identifier is a string, a switch statement will not work. "Identifiers" is a lookup class that contains a list of static constant strings to compare the identifier against.

For example, if we find a more complex element like an attribute or sequence, we can call the "addChild" function to add the element to the child. The constructor of the element being added as a child will then process the XSD representation of that element, creating a recursive building structure that builds the entire data structure from the XSD.

7.2.1 Constraints

Within XSD one can impose extra constraints on the datatype contained within an XML element. The following constraints are available within XSD:

Constraint	Purpose
enumeration	A list of values that a datatype can obtain.
fractionDigits	Amonut of decimal places allowed.
length	Either exact amount of characters in string, or exact amount of items in list.
maxExclusive	Maximum value of numbers, value itself not included.
maxInclusive	Maximum value of numbers, value itself included.
maxLength	Either maximum amount of characters in string, or maximum amount of items in list.
minExclusive	Minimum value of numbers, value itself not included.
minInclusive	Minimum value of numbers, value itself included.
minLength	Either minimum amount of characters in string, or minimum amount of items in list.
pattern	Regular expression that a string must follow.
totalDigits	Exact amount of allowed digits.
whiteSpace	What the datatype is allowed to do with whitespaces.

Figure 68: All constraints available within XSD

These, however, present a problem for the parser generator, as the goal of the parser generator is to generate a grammar that checks whether a format is valid and then processes the data contained within that format. Checking for constraints like these is, however, highly problematic, as many of these checks are not possible by a parser when one follows the default definition of a parser. For example, parsers generally cannot check whether a number is within a specific range, as the parser operates on text. In order to support constraints like this, these constraints would need to be added as explicit features to the parser builder. This issue is resolved in two ways.

The first is by exporting all constraints. Besides generating the UBNF grammar, the converter exports a list of elements together with their constraint. This allows one to develop tools that check whether constraints are appropriately adhered to and append such tools to the parsers.

The second, more practical solution is that a large part of the constraints were added to the parser builder. We added language support for nine new UBNF syntax features that allows one to impose constraints on grammar rules. These constraints are automatically generated by the XSD to UBNF converter and inserted into the relevant grammar. This covers all the length and value constraints possible within XSD, where the other constraints can be checked in post-processing using the exported list of elements and their constraints.

7.2.2 Substitution

Substitution is a concept within XSD that allows one element to directly replace another element. One common use case for this feature is to allow the XML elements to be defined in multiple languages, by substituting the name of an element by its same name in another language. The following XSD exemplifies the workings of substitution:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xs:element name="user">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="name" type="xs:string"/>
<xs:element name="naam" substitutionGroup="name"/>

</xsd:schema>
```

Figure 69: An XSD example using substitution.

```
<user>
    <name>FirstName</name>
    <naam>SecondName</naam>
</user>
```

Figure 70: A valid XML file according to the XSD.

As can be seen, the core of the XML is a "user" element, which contains any amount of name elements. A name can, however, be substituted by a "naam" element, the Dutch word for name. If the substituting element defines no type, its attributes and children are precisely equal to the substituted element. In contrast, if it does define a type, it must inherit from the substituted element, meaning that all elements contained within the substituted element must also be present in the substituting element.

This inheritance, where the substituting element adds extra content to a substituted element, makes generating a UBNF grammar much more complex. If a substituting element could not define a type, it would simply be a matter of making the name of the substituted XML element, which is located in the opening and closing tags of the element, an OR sequence of all the possible names the element can have. So, in this case, the name would be "naam | name", where | indicates that it is either one or the other. However, this implementation does not play well with inheritance.

The solution that eventually was implemented was to generate a separate element definition of each substituting element. So, in this case, both "name" and "naam" would be defined entirely within the UBNF. This implementation may sound inefficient. However, since we are guaranteed that all content contained within the substituted element is also contained within the substituting element, those two elements can share the definitions of that content, requiring the attributes and children of the substituted element only to be defined once.

7.3 Class structure

The following image shows the class structure of the XSD to UBNF converter:

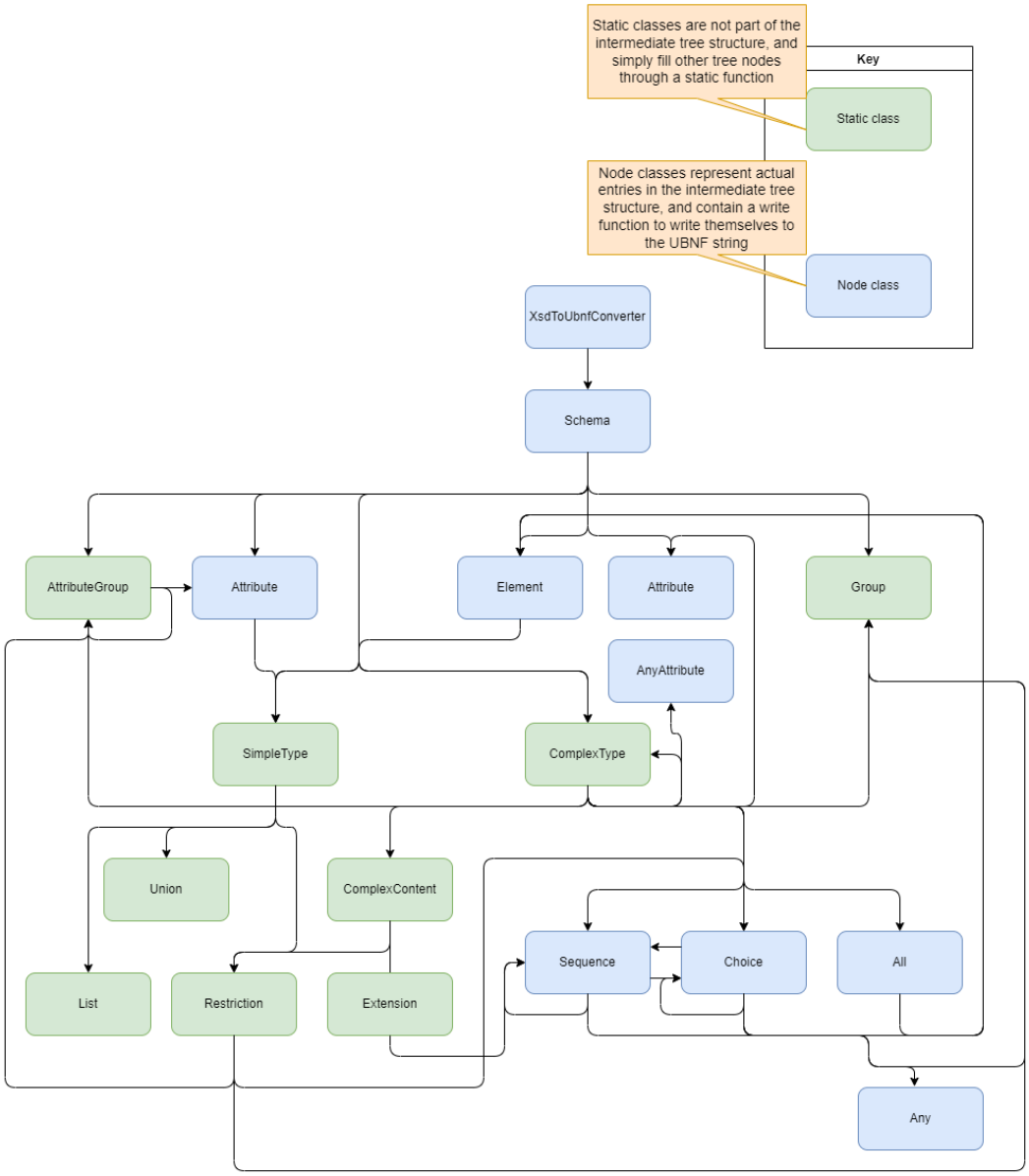

Figure 71: The class structure of the converter.

This diagram represents which classes call other classes. Note that this is a simplified overview of the class structure. One example detail that has been left out is inherited classes. This choice was simply made because the diagram would become too convoluted and unreadable, leading to less information obtained by the reader, even though the diagram contains more information. As such, please understand that this is a high-level overview of the architecture, with many details aggregated or left out.

7.4 Post-processor

Assembling the data structure that can be written to UBNF is a complex task due to inheritance being a supported feature in XSD. What this means is that within XSD, when one specifies the content of an XML element, it can not only define this content as children of the XSD element, but also outside of the XSD element, which is then referenced by the XSD element through its type attribute. The following figure exemplifies this:

```
<xs:element name="shiporder" type="orderType"/>

<xs:complexType name="orderType">
  <xs:sequence>
    <xs:element ref="orderperson"/>
    <xs:element ref="shipto"/>
    <xs:element ref="item" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="orderid" use="required"/>
</xs:complexType>
```

Figure 72: Example of inheritance within XSD

Note how the element references content defined outside of the element itself. Also note that even though it is not displayed within this example, the type defined outside is also allowed to then inherit other content. This would normally not be a very significant issue, but the issue is that such content is allowed to be defined either before or after the element's definition. This means that by the time that you encounter the element, you may not have encountered the type yet, and therefore you are not able to fully construct the element at that point in time.

The solution is a post-processor, which goes over the elements generated so far, and adds content discovered later. This requries us to keep track of two things:

1. A list of requests, which keep track of elements that request to be extended, together with the name of the types that need to be added.
2. A lookup table of discovered types, who's lookup keys are the names of the types, and who's entry content is the data contained within the type.

The post-processor then simply has to match the requests with the types to achieve the final result. One further complexity that, however, needs to be taken into account is the fact that types available in the lookup table may themselves be requesting data to be appended to them. This can be solved by ensuring that whenever the lookup table entry is used, the requester of that entry becomes the requester of all requests that refer to the lookup table entry. The following diagram shows this process:

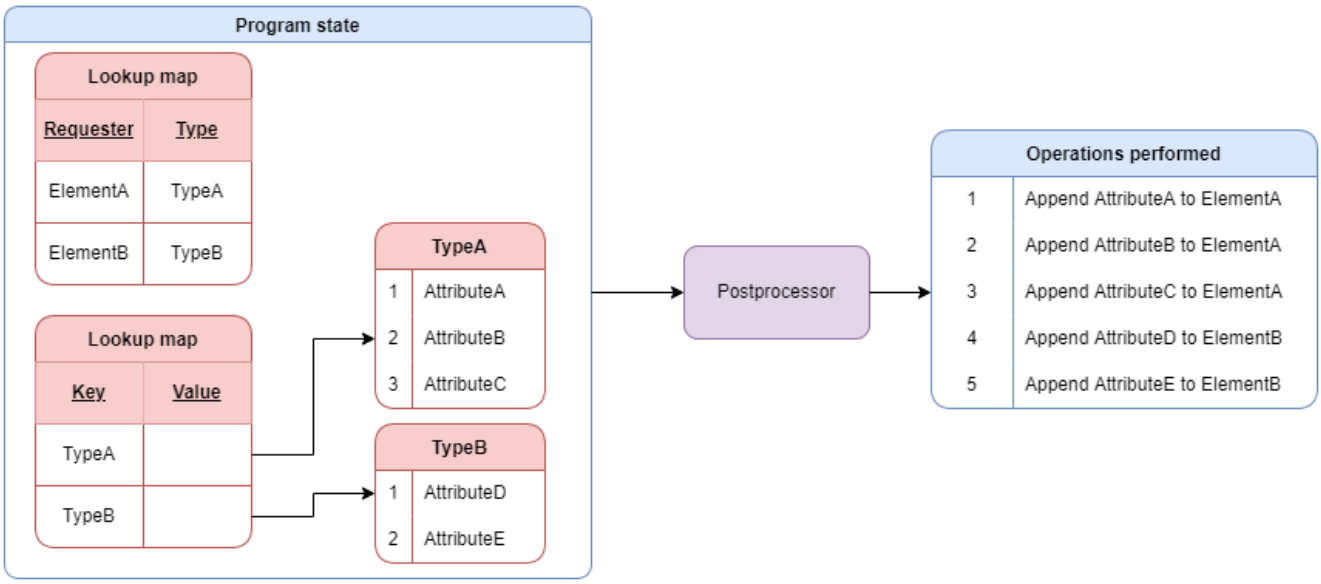

Figure 73: The post-processor's workings.

7.4.1 Element types

So far, what has been shown is that types can be inherited. Elements can however also be inherited. XSD refers to this process as referencing, where an element can reference another element through the "ref" attribute, which essentially works the exact same as the "type" attribute does for inheriting a type. Handling this is not the most difficult task, but there are a few details to consider with this process.

As referencing works nearly the same as inheriting does, the same post-processor can be used for most tasks. It is simply a matter of adding the reference requests to the post-processor's requests, and adding the elements' contents to the list of available types. There are however two problems with this.

The first problem is that most elements are actually not referred to, and simply directly used. It is extremely inefficient to store every element's content as a possible type that can be referred to. The XSD specification however specifies that only root elements can be referenced [5]. Most XSD files contain very few root elements, and XSD files that contain a lot of root elements often use them for referencing. As such, storing the data of root elements is a perfectly reasonable design decision for the post-processor.

The second issue is that types and elements are now inserted into the same pool of possible types. While two elements cannot be defined with the same name as root, it is possible to define both a type and an element of the same name, which is easy to distinguish in XSD, but its difference gets lost when inserting into the pool of available types. To remedy this, all types' names are appended by "Type", while all Elements are appended with "Element".

7.4.2 Common references

It is, however, not as easy as just matching up types. Multiple elements can reference the same type. If we were to add the type to each element, a type that may only be defined once in the XSD would be defined for every element in the UBNF. This is inefficient because one can instead create a common type referenced by multiple elements, reducing the size of the final parser.

One idea to resolve this problem was to make the post-processor check whether an element is used multiple times and, if so, create a common type for all elements to reference. A more efficient idea, however, was to give each type a common name and then pass this common name to all elements inheriting the type. The writing stage of the XSD to UBNF converter was written in a way that it does not write a rule with the same name twice. So if multiple elements contain a rule with the same name, it will only be written once, and then all elements who use a rule with the same name will be able to utilise the same grammar rule.

7.5 Output

An example of the conversion process can be seen in the appendices. Appendix B gives an example of an XSD in string form (the converter would take it as a Schut array, but these cannot be easily displayed within this document). Appendix C then gives the output of the converter for this specific case.

8 Analysis

8.1 Summary

This section concerns itself with the analysis phase of the project. This analysis phase consists of two parts: Finding limitations within the company's toolset and analysing the performance and accuracy of the parser generator created during this project. The toolset limitations will be discussed in section 8.2, while the parser generator's performance and accuracy will be analysed in section 8.3. The performance will be analysed by giving a detailed look at the performance of the individual components of the parser generator on a set of wildly different XSD projects. In constrast, accuracy will be analysed by comparing the accuracy of the parser generator against other XSD tools.

8.2 Toolset limitations

This section will concern itself with limitations found within the tools employed by Schut. It serves the purpose of performing research for Schut on which systems need improvement. Some of these limitations may have been solved during the project, while others may not. A significant part of the project is to analyse which features are missing. Some of these issues are too complex to resolve within the project, so an unresolved analysis subsection does not indicate that the project was not fully completed. Anything that was not completed was never intended to be fixed during the project.

8.2.1 Parser builder

This section describes discovered issues within the parser builder, which is the IDE used by Schut to work with UBNF, with which our tool is integrated. It allows syntax-highlighted writing of the UBNF grammar and contains the build feature used to build the state machine that matches the UBNF grammar. One can also run this state machine on a string matching the UBNF grammar and export the parser-generated Schut array..

8.2.1.1 UTF encodings **Resolved (Section 5.3.3)**

There are many different methods to encode text. Two frequently used options are UTF-8 and UTF-16. They are essentially similar, but may use a different amount of bytes for a specific character, leading to different byte sequences for the same piece of text. As such, they are not directly compatible. Schut uses UTF-16 internally within their tools but may decide to use UTF-8 for input or output. Generally, the input should however be able to detect whether the input is UTF-8 or UTF-16, and change the input encoder. The parser builder however does not do this, and as such, all file import options strictly require UTF-8. This lack of checking means that many files are incompatible with the tool.

The tool's import functionality should use a stream that can detect common text-encodings, and change the stream's encoder to match the encoding used by the input file.

8.2.1.2 Error messages **Resolved (Section 5.3.1)**

The error messages produced by the parser run within the parser builder were significantly limited. It hampered the testing workflow when attempting to write the grammar for the XSD parser. Contrary to most software development projects, parsers can make very counter-intuitive decisions, with examples being a very unrelated token being matched while resolving a grammar rule, due to the token being the longer one, or the parser going into an unexpected production rule. Proper understanding by the programmer of a grammar they just wrote can only be achieved if the parser builder produces detailed explanation that not only describes that there was an issue, but exactly why it ran into the issue, and what the conditions would have been to not run into the issue.

Prior to this project, the error handler purely reported that it was not able to parse a string, together with

the exact location within the string that was not parsable. This error message misses a few crucial points to make a user understand the error:

- Which production rule is it currently trying to resolve.
- Which token did the parser form when it failed.
- Which list of tokens did the parser consider following valid tokens to continue the parsing process.

8.2.2 Generated parsers

This section describes discovered issues within the generated parsers, which essentially discusses issues with how to generate and interpret the state machines.

8.2.2.1 Token contamination **Resolved (Section 5.3.2)**

Arguably, conflicts are the greatest challenge of building parsers. Conflicts arise when the parser does not have one clear next step, and the parser has to choose between multiple possible routes. While the programmer is often presented with errors or warnings, it may still be possible to build the parser, but the parser may make a choice that the programmer did not intend.

There are two types of conflicts, token conflicts and grammar conflicts. Token conflicts arise when the string that needs to be parsed can be tokenised in multiple ways. For example, let us take the following small bit of code:

```
/*
This file contains a lot of comments
*/
int main() {
  return 0;
}
/*
Here is another comment
*/
```

Figure 74: A small piece of C code.

The individual tokens this file contains may seem trivial. However, suppose one defines a comment as being opened with "/*" and closed with "*/". In that case, it is not unreasonable for the parser to match the comment opener on the first line with the comment closer on the last line, making the entire file one large comment. Moreover, without any special care, most parsers will do precisely that, as a parser always tries to take the longest token possible.

Grammar conflicts are when the parser does not know which rule to apply next, as it has a choice between multiple rules. There is a trade-off when designing a parser builder between how quickly it runs into conflicts and how fast the resulting parser is, as one can create smarter parsers that can look farther ahead in the string, but they take longer to execute.

Usually, these two conflicts are pretty manageable. One defines the list of tokens in a separate file, often used by the tokeniser, making it easy to compare tokens for any conflicts. At the same time, the grammar rules applications are a matter of very localised issues that are pretty quick to track down. However, UBNF makes this process more complicated.

The tokens are defined in line within the grammar rules of UBNF. One can specify a string to match in the middle of a grammar rule, which adds that string to the list of tokens. As the token matching is however independent of the tree structure for the grammar rules, adding an extra token to a grammar rule can break another token in an entirely unrelated grammar rule. This means that one is at any point at risk of breaking a very unrelated part of the grammar without having a proper overview of the list of tokens. This is referred to within this research as token contamination.

Let us take the following UBNF example:

```
<Root> := <A> | <B>;

<A> := "x" "s" ":element";
<B> := "xs:" "schema";
```

Figure 75: An example of a grammar that does not behave as expected.

One could easily see this is trying to match two strings: "xs:element" and "xs:schema". However, if ones tries to provide it the string "xs:element', it will reject it, stating that it expected "schema":

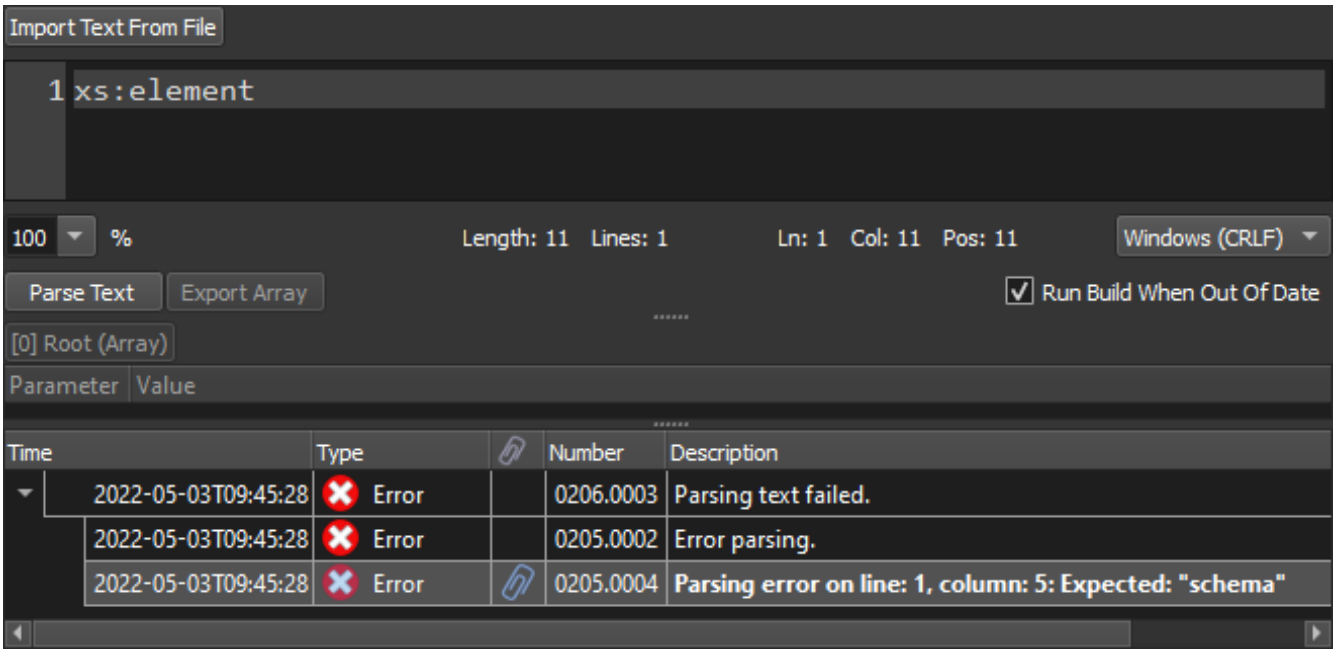

Figure 76: The output for the string "xs:element".

This result is produced because the tokens it tries to match are independent of the grammar rule it is currently in. For xs:element, there are two tokens it can immediately match when taking the list of all our tokens, "x", and "xs:". As it always tries to match the longest token, it will take "xs:" as the first token. It can then only find one grammar rule that starts with "xs:", and tries to complete that grammar rule, which it cannot. This example is quite artificial and easily fixable, but these kinds of issues can appear quickly in large grammars.

8.2.2.2 Empty production rules — Unresolved

Another issue contained within the parser generator is the fact that it cannot deal appropriately with empty rules. The language itself does not accept an empty rule, and it cannot handle the evaluation of a rule with optional components to nothing. This means that if a rule contains only optional components, it expects at least 1 of the components to appear, even though all components were specified as optional. The example used to research the behaviour of the tool regarding this problematic behaviour will be focused on the following small example:

<Failure> := ”a” <B>;
<B> := ”b”?;

Figure 77: An example of a grammar that does not behave as expected.

The parser generator performs a poor conversion at the very first step of the conversion process. The following is the original, non-processed state machine for this grammar, followed by the result of the first processing step:

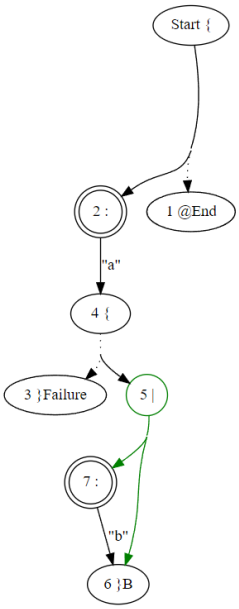

Figure 78: The non-processed implementation of the grammar.

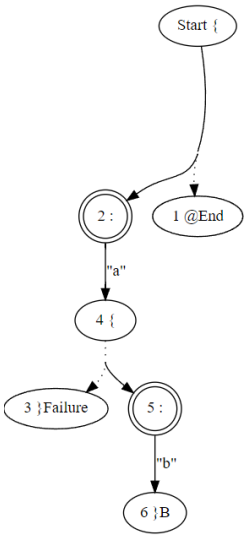

Figure 79: The implementation of the grammar, processed one time.

The parser generator has 1 OR transition to resolve, the one at node 5, which is the OR between using a b or not using a b, which describes the optional nature of the terminal. Clearly it however does not take the proper step here, as it resolves away the option to transition to the end of the rule without actually reading a ”b”.

The reason that this breaks is due to an assumption made during the optimisation process. The parser builder assumes that grammar rules can never produce an empty Schut array as the result, and as such it thinks it is safe to make this optimisation. Unfortunately this is not a simple thing to fix, as the optimisation is not there to just shrink the size of the graph, but to resolve conflicts.

When one would allow empty production rules, situations appear where there are multiple correct routes, and

the parser would not know which to take. For example, let's say a rule that evaluated to an empty set is itself an optional rule. Then going into a rule, producing an empty Schut array, and exiting the rule is an allowed path. But not going into the rule is also an allowed path. Both routes will not read any tokens from the input, but produce different output (appending an empty Schut array vs not appending anything), and it is therefore impossible to decide what the output is.

It is possible to support such a feature, but will take very careful consideration of what is allowed syntax within UBNF, and how to process all cases. This is too extensive of a project to take as a side project within building the parser generator, and can therefore not be fixed with current resources.

8.2.2.3 Control characters **Unresolved**

As UBNF allows one to create a parser generating a Schut array, it naturally has a syntax that allows the grammar designer to modify how the Schut array is generated. One very natural operation is telling the parser not to include a token in the Schut array. This is specified through the "~" character. As such, the following would be a valid UBNF specification:

```
<S> := "a" "b" ~"c" "d" ~"e";
```

Figure 80: Example of a grammar using the control character "~".

As "c" and "e" got a control character in front of them, these 2 symbols are expected in the input, but will not be stored in the output array. The array produced by the string "abcde" for this parser would be:

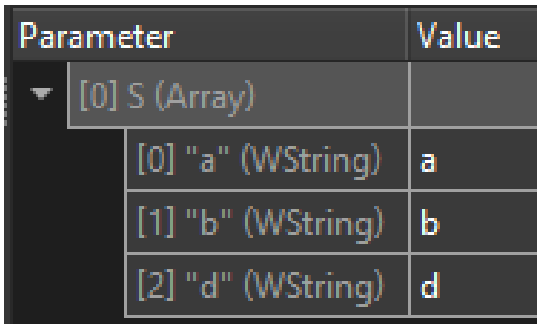

Figure 81: Output array of the grammar.

This system however introduces an unexpected yet highly problematic bug. This control character should purely influence the output of the Schut array, and not the set of accepted strings, yet it does. Because adding the control character changes the name of the token. Internally, the system sees "a" and ~"a" as two entirely different tokens, yet it can match both tokens using the same character, in this case a single "a". So if one defines the production rules in such a way that at some point in time the parser can choose between matching "a" and ~"a", it will always match one of them, and try to resolve the production rule belonging to the choice made, and completely ignores the other one. As such, for the following grammar:

```
<S> := <A> | <B>;
<A> := "a" "b";
<B> := ~"a" "c";
```

Figure 82: Example of a broken grammar.

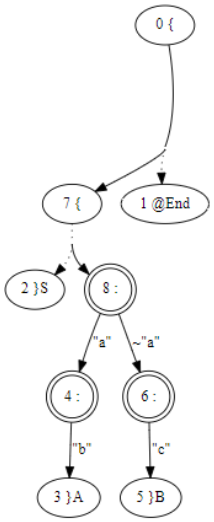

Figure 83: The corresponding state machine of the grammar.

The system will be able to accept the input "ab", but it will reject the input "ac", simply because it will always go into the first matched token, which is "a", as the A production rule is defined above the B production rule, and ignores ~"a" because it only explored multiple routes if the matched token is the same, and ~"a" is seen as a different token compared to "a", as seen in the state machine.

8.2.2.4 Grammar parameters

Resolved (Section 5.3.5)

Grammar rules within UBNF can contain parameters. The following is an example of a grammar with parameters:

```
<Root> := <a> <b>;
<a; ArrayId="aRule"> := "a";
<b; ArrayId="aRule"> := "b";
```

Figure 84: Example of a grammar using parameters.

Here both rules contain the "ArrayId" parameter, which instructs the parser to give the output array representing the rule the name specified in the parameter. These parameters are all stored within the state machine data structure at the end of a grammar rules, so the grammar parameters can be processed once a rule has been matched. This works well, except for one issue: The system will always attach all parameters to every grammar rule, no matter if they are used or not. They are simply left empty or set to a default value when unused. This leads to the exported parsers' file size being unnecessary large, the program using more memory than necessary, and access time to the end rule nodes being longer than necessary, as there is more data to search through. A more optimal system would be a system that would only add parameters if they are actually defined.

8.2.3 UBNF

This section describes discovered issues within UBNF, which essentially discusses missing features from the UBNF syntax.

8.2.3.1 Bounded repetition — Unresolved

Repetition is a critical part within regular expressions. The most common ways to indicate repetitions are through the kleene star operator (*), which indicates an element is allowed to repeat anywhere between 0 and infinite times, while the + operator indicates repetition between 1 and an infinite amount of times. However, one other case that users of regular expressions may want to specify is a bounded repetition, where there is either a minimum amount of repetitions different from 0 and 1, or a maximum amount of repetitions different from infinity. For example, one may want to specify that an element is allowed to repeat between 4 and 6 times.

Within the theoretical model of regular languages, which is the mathematical basis of regular expressions, this is not supported, as, for example, 4 to 6 cases can be simulated by 4 mandatory elements followed by 2 optional elements of the same type. This is however not a nice way to write such regular expressions down, so computer-related regular expression languages often support bounded repetition in some way or other. This is notably lacking from the regular expressions supported in UBNF. The lack of this feature is interesting within this project, as bounded repetition is a handy feature. Within XSD, one can specify the minimum and maximum occurrences of an attribute or element, which could be directly mapped to a bounded repetition. Due to the lack of this feature, one would however need to generate x mandatory occurrences of an element, followed by x optional occurrences of an element. The following is an example of this:

```
<xs:element name="shipto">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="location"
            type="xs:string" minOccurs="4"
            maxOccurs="6"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 85: An XSD example with bounded repetition.

```
<shipto> :=
"<shipto>"
<location> <location>
<location> <location>
<location>? <location>?
"</shipto>";
```

Figure 86: The unnecessarily complex UBNF grammar if bounded repetition is not supported.

8.2.3.2 Non-greedy operator — Unresolved

One of the more significant challenges within tokenising is matching the shortest token, rather than the longest. This was discussed during section 8.2.2.1. While matching the longest token is often the intention, there are cases where one wants to deliberately match the shortest token, as shown in figure 74. Within many implementations of regular expressions, a special character combination exists referred to as a "Non-greedy operator". IBM is one such company that implemented such an operator into their tools [36].

Essentially, when any control character that can modify the length of the token is proceeded by a "?" character, it will try to match as few tokens as possible rather than as many as possible. As such, "??", "*?" and "+?" will all first try to match 0 tokens, and then incrementally add tokens until they find the first case where they can match a token. Support for such an operation would be beneficial to the tool, as it is incredibly tough to implement a regular expression for matching the shortest possible token without such an operator.

8.2.3.3 Static array additions — Resolved (Section 5.3.4)

One interesting feature that was noticeably lacking was the idea of static array additions. A static array addition is an act of adding an element to a Schut array independently of what is written in the text being parsed. Essentially, it is the ability to add data to a Schut array that is defined in the grammar rather than in the file being parsed.

This may not seem like a handy feature, but the idea of the relation between the generated parsers and their output array is that the data contained within the file being parsed is included in the array. However, it is quite beneficial. To exemplify the use cases based on the parser generator written during this project: XSD defines a type for data. For example, an element may contain an integer, a double, a positive integer, or any other data type commonly encountered in computing science. However, this information is lost when the UBNF grammar is generated, as it simply generates a regular expression matching the data type's format. But an integer and a short for example have the same regular expression defining it. As such, it is helpful to add the data type contained within the XSD element to the content field of the resulting Schut array, so the user of the array is aware of the data type of the data.

In essence, it would be useful to have the ability within UBNF to insert data into a Schut array without the data being necessarily there in the file being parsed.

8.3 Performance and accuracy

This section will concern itself with the performance of the tool crafted during this project, including its performance, development time, advantages, disadvantages and limitations.

To analyse the performance of the parser generator developed within this project, we will be looking at a set of XSD projects that are to be converted to a parser. For each XSD project, a datasheet will be provided that shows both the details about the project being processed, and the resulting performance metrics.

Beyond the metrics specific to the parser generator developed during this project, a comparison will be made on the accuracy and performance scale with comparable XSD projects. Unfortunately, to the best of our knowledge, no other parser generator exists right now that can generate parsers from XSD that perform a data operation, and most projects are focused on schema compliance verification. As such, there will be a difference in tasks performed by the compared projects, which may lead to poor performance comparison. This detail will be further worked out in the discussion, however.

8.3.1 Test setup

This section gives a detailed report of the resources used to perform the analysis. It describes the XSD projects used for analysis, the tools that are compared against the parser generator, and the hardware that the tests are performed on.

8.3.1.1 XSD projects

The following repository contains all projects used in both performance analysis of the parser generator, and in the analysis with other projects:
https://github.com/LarsSven/Thesis/tree/master/Test_Projects

All projects were chosen to be used for analysis based on their characteristics and them being interesting and possibly difficult tests for any program trying to use them. None of these projects give the parser generator an advantage in performance or comparison to other tools, as they were not tested against the parser generator prior to choosing to use them for analysis. The following table displays information about each project:

XSD project	Comments
Purchase Order	A classic XSD project. It is a simple schema that tests the basics of XSD. It was created by the developers of the XSD schema, and is used as an example when teaching programmers XSD.
IBM Substitution	A test project provided by IBM to developers learning XSD. It has no real-world application, but is special because it makes use of substitution, which is a rare and hard to support feature within XSD.
IBM TestSuite	A reasonably simple format used by IBM to export results of their unit test framework "Code review" to XML. Makes use of lots of inheritance, which is always a challenge, but it does not utilise anything too difficult to support.
SGM	A format that is frequently used by Schut. It provides the XML format for exporting Schut arrays. It is of a similar difficulty to IBM testsuite, though it is a bit larger in size.
Thales Web Service	A format used by Thales to communicate with its web services through XML requests. It provides a significant amount of restrictions within its format, which truly tests the capabilities of dealing with restrictions properly within XSD.
Thales Activation	Another format used by Thales to communicate with its web services. Its purpose is related to generation and usage of protection keys. It is of similar difficulty as Thales Web Service, but is significantly larger in size.
SEPA	SEPA is a European payment transfer system. It is used by practically every bank within Europe, and the SEPA XSD describes the XML format necessary to work with this system. Its main difficulty is the enormous size of the Schema, leading to longer execution time, and simply much more elements presented where issues can occur during parsing and processing.
UBL	UBL is another banking system. Invoices can be created through the XML schema. The UBL XSD schema is of an absolutely enormous scale, leading to very serious problems for any system not being able to deal with an arbitrary XSD file. Beyond testing the absolute limits of large XSD processing, it spreads its schema through multiple files, and then makes use of XSD's import and include feature. This is a somewhat uncommon feature, and not supported by all XSD tools.

Figure 87: The XSD projects that will be used for the analysis of the parser generator's performance and accuracy.

More detailed metrics of the exact size of the projects is described in the section 8.3.2, as this data is attached to the actual performance results.

8.3.1.2 XSD tools

This section will describe the set of tools that will be used to compare the accuracy of the parser generator with. Unfortunately, as there are no known cases to us of tools that allow one to generate parsers from XSD that perform more complex tasks than Schema validation, it is impossible to compare the performance of the tools with the parser generator, as the parser generator performs a task that is significantly more complex.

Due to the fact that the parser generator is significantly more complex, the parser generator will be orders of magnitude slower than the tools it is compared against, as they perform simple tasks like converting from one schema to another, or generating code representing the Schema. These tools will therefore only be used to compare the accuracy against the parser generator. The following tools will be used for accuracy analysis:

Tool	Version	Author	Comments
XSD to C#	4.6.1055.0	Microsoft	A tool to convert XSD files to C# classes in order to work with the Schema in a programmatic manner. It requires code to be appended to the C# classes in order to be able to do anything with the data.
XSD to JSON	2022.1.3	Syncrosoft	A plugin contained in many IDE's that allows one to convert an XML schema definition to a JSON schema definition. For this research, the IntelliJ IDEA plugin has been used.
XSD Validator	0.0.3	Mike Kroutikov	A python tool for checking whether an XSD file adheres to a specific XML Schema Definition.
JAXB	3.0.0	Oracle	A Java tool that converts XSD files to Java classes that represent the XML format.

Figure 88: The tools that will be compared against the parser generator.

8.3.1.3 Hardware

In order to guarantee reproducability, a list of hardware used during testing is included:

- **CPU:** Intel Core i7 7700K
- **GPU:** NVIDIA Quadro K420
- **DRAM:** 16GB DDR4 2400MHz
- **Storage:** Samsung 750 EVO 500GB
- **Operating System:** Windows 10 Pro 64-bit

8.3.2 Performance

The average results of 5 rounds of performance testing will be presented per XSD project. While this section will only show the aggregated results, appendix D shows the individual results on a per-run basis. For each project, a figure will be presented that consists of two areas:

1. Project details
2. Performance details

The project details will discuss details about the project being tested that is relevant to someone intending to interpret the performance results. Information within the project details include information like the size of the XSD project (Amount of lines of XSD, amount of files, amount of root elements), data size of the XSD files, and information such as how many types and how many elements are present within the project.

The performance details will then give details on the performance of the parser. It will provide the execution time of every component of the full parser generator, combined with the total execution time of the entire pipeline. It will also provide how many types and how many type requests were generated by the post-processor to handle.

Note that the execution times values will be presented in milliseconds, rounded to 1 decimal. As such, if the execution time of a part of the process is presented as 0 milliseconds, it took less than 0.05 milliseconds to execute.

Purchase Order			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
66	1	2	
Amount of types:	Amount of elements:	Total size of XSD files:	
4	17	2.31 KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	8.7 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
1565.4 ms	1035.9 ms	2611.9 ms	
Amount of type requests:	Amount of available types:		
7	6		

Figure 89: Purchase Order results.

IBM Substitution			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
65	1	7	
Amount of types:	Amount of elements:	Total size of XSD files:	
5	16	1.88KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	9.5 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
1171.8 ms	737.7 ms	1924.5 ms	
Amount of type requests:	Amount of available types:		
8	11		

Figure 90: IBM Substitution results.

IBM TestSuite			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
65	1	1	
Amount of types:	Amount of elements:	Total size of XSD files:	
8	8	2.63KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	11.5 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
1537.6 ms	979.9 ms	2529.1 ms	
Amount of type requests:	Amount of available types:		
8	8		

Figure 91: IBM TestSuite results.

SGM			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
331	1	1	
Amount of types:	Amount of elements:	Total size of XSD files:	
37	33	11.4KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	33 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
2538.9 ms	2945.7 ms	5518.1 ms	
Amount of type requests:	Amount of available types:		
72	41		

Figure 92: SGM results.

Thales Web Service			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
421	1	1	
Amount of types:	Amount of elements:	Total size of XSD files:	
47	91	16.7KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	48.9 ms	0.5 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
26337 ms	9626.2 ms	36012.6 ms	
Amount of type requests:	Amount of available types:		
14	14		

Figure 93: Thales Web Service results.

Thales Activation			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
373	1	1	
Amount of types:	Amount of elements:	Total size of XSD files:	
86	128	15.8KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	39.7 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
2408.6 ms	1836.4 ms	4285.2 ms	
Amount of type requests:	Amount of available types:		
26	11		

Figure 94: Thales Activation results.

SEPA			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
2371	1	1	
Amount of types:	Amount of elements:	Total size of XSD files:	
109	210	130KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	103.1 ms	1.6 ms	0.4 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
4932.1 ms	5610 ms	10647.1 ms	
Amount of type requests:		Amount of available types:	
217		109	

Figure 95: SEPA results.

UBL			
Project details			
Total lines of XSD:	Amount of files:	Amount of root elements:	
49058	15	1621	
Amount of types:	Amount of elements:	Total size of XSD files:	
1197	3912	2710KB	
Performance details			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1751.6 ms	18.4 ms	18.8 ms	16.1 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
2822960.2 ms	1323955.9 ms	4148426.1 ms	
Amount of type requests:		Amount of available types:	
4842		2801	

Figure 96: UBL results.

If these results are plotted on a scatterplot, and a line of best fit is drawn through it, the following result appears:

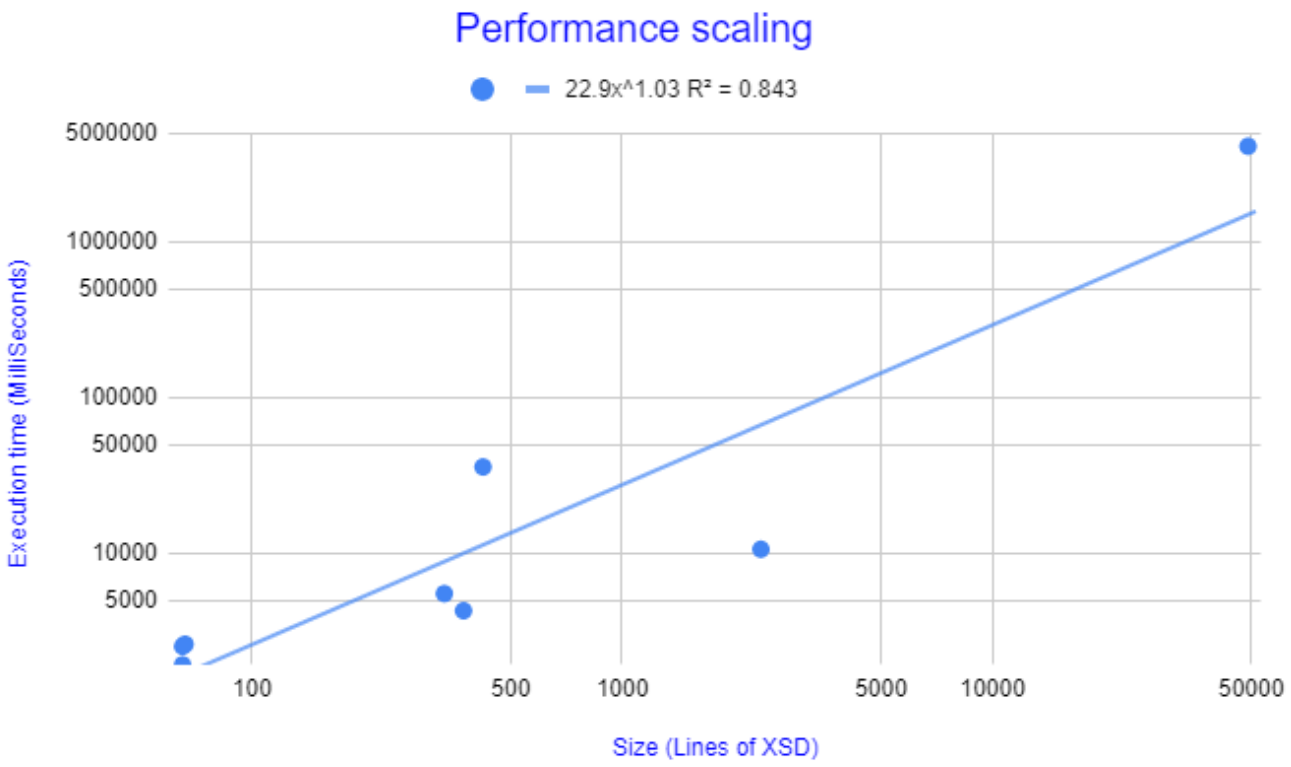

Figure 97: A scatterplot of the parser generator performance results.

Note that the trendline is a power series, which was chosen as it visually best matches the dataset. Furthermore, note that both the horizontal and vertical axes are logarithmic, which is the only way to visualise this dataset, due to its wide range of amount of XSD lines.

The first thing to realise about the performance results is that the results are highly acceptable. Nearly all projects finish in a perfectly reasonable time for a user to wait, the only exception being UBL. For an average project, which refers to the smaller projects, XSD projects are generally not that big, so a user only has to wait for a few seconds. Waiting this long would, for everyday usage, already be highly acceptable. However, one also has to consider that this is to generate a parser, which is a one-time operation to create a software package. People performing such tasks are often highly comfortable with longer wait times, which means that a few seconds is actually really fast.

UBL takes nearly an hour and a half to complete. The reasoning for waiting this long is a combination of two factors: First, it is simply an extensive project, and second, it is somewhat poorly designed. UBL is the only multi-file project being analysed, and it does some interesting tricks through the other files. What the project essentially does is define library files that contain element definitions as roots. The main file then references these definitions. While this works, this has an issue: All root elements can be starting points of an XML file. As such, every single one of the library element definitions is considered a starting element, which is, of course, not intended. This design feature leads to a highly complex parser. UBL tries to resolve this by annotating the actual root element with a

statement that that element should be the root element, but this is not programmatically interpretable, so it can only be resolved by explicitly asking the user which elements should be considered root elements.

There are two outliers in our performance results, which become especially apparent in figure 97. These are the Thales Web Services project, which is much slower than expected, and SEPA, which is much quicker than expected. Thales Web Services takes such a long time because its structure is inherently very complex and compact, leading to the amount of grammar being necessary to describe a small part of the XSD being a lot more. This leads to a much larger final parser, which takes much longer to generate. SEPA is much quicker because it primarily consists of annotations, which are sections of the XSD that require practically no translation, leading to the actual final parser size being very small compared to the size of the XSD.

While the performance of the parser generator itself may be interesting, the parser generator, of course, generates parsers. These parsers themselves also have a performance profile which can be analysed. The performance of the parsers themselves is arguably even more important than the performance of the parser generator, so this deserves to be analysed as well. The following data shows the performance of each individually generated parser on a an example file. All example files are located in the test projects repository mentioned at the start of section 8.3.1.1. All results are an average of 5 repetitions.

Generated parser	Test file	Test file size	Execution time
Purchase Order	Examples/Purchase_Order.XML	35 lines	2 ms
IBM Substitution	Examples/Example.xml	23 lines	1.3 ms
IBM Testsuite	Examples/Example.xml	10 lines	1.4 ms
SGM	Examples/sgm_xml_test_ArrayWriterTest _SimpleWriteTest.xml	380 lines	23.3 ms
Thales Web Service	Examples/Example.xml	231 lines	14.9 ms
Thales Activation	Examples/ThalesActivation.xml	17 lines	1.6 ms
SEPA	Examples/Rabobank_Example.xml	141 lines	9.2 ms
UBL	Examples/PricingExchangeRate.xml	103 lines	10.4 ms

Figure 98: Performance of the generated parsers on example files.

Plotting this on a scatterplot and drawing a trendline gives the following graph:

Figure 99: A scatterplot of the parser performance results.

This data is by no means extraordinary, and it contains no interesting skews or other unexpected oddities. It does however allow one to draw a few interesting conclusions about the performance of the generated parsers that are both highly positive, and rather surprising. There are two big takeaways from the data:

1. The relationship between the parsed file's size and the parser's execution time is linear.
2. The size of the generated parser has a negligible effect on the generated parser's performance

This first point leads us to conclude that the parser is fast, with an excellent performance curve. It can efficiently deal with larger XML files, leading to performance that perfectly competes with manually crafted parsers. The second point is interesting: No matter how large the XML format is, it will have an extremely tiny effect on the parser's execution time. That second point is a terrific attribute for the parser generator, as one can craft a more extensive XML specification without worrying about the resulting parser's performance.

8.3.3 Accuracy

8.3.3.1 Conditions

The accuracy is determined by running every tool on the 8 projects introduced in section 8.3.1.1. The conditions for a project to be accepted by a tool is determined on a per-project basis based on the result a user may expected from a tool.

One global condition that every project must always at least meet for the XSD project to be considered accepted is the following:

- The XSD was fully parsed without any errors.

Besides this global condition, specific conditions were selected for each project. While these unfortunately do not represent the full projects being parsed, a condition along the lines of "The output should be the expected result" is not reproducible nor stable. As such, a few interesting points are selected from each project in order to get stable conditions for each project. Note that these conditions were not compared against any of the tools prior to being selected. The following conditions apply per project:

Purchase order:

- Both the "purchaseOrder" and "comment" elements are allowed as root elements.
- "purchaseOrder" can have 1 attribute: "orderDate".
- "purchaseOrder" can have 4 children: "shipTo", "billTo", "comment" and "items".

IBM Substitution:

- "LiteratureStore" exists, and can contain the exact same info as "BookStore" (Tests for properly supporting substitution groups).
- "Book" exists and contains all information provided in "BookType" (Tests for properly supporting substitution groups when they also have their own inherited type).
- "MyString"'s restriction of being between 3 and 10 characters long is properly handled.

IBM Testsuite:

- "files" is an optional child of "report".
- "historyId" is a required attribute of "report".
- "rule" and "category" can appear between 0 and an infinite amount of times as a child to "category".

SGM:

- "typeInfo" contains 2 optional attributes: "qty" and "uid".
- "referenceValue" contains 3 children: "base", "item" and "type".
- "doubleVectorType" contains 1 child: "dval", which can occur between 1 and an infinite amount of times.

Thales Web Service:

- "productName" is a child of "product", must appear exactly once, and is a string of a length between 1 and 50.
- "memorySegment" contains a child "name", which is allowed to be empty.
- "attributeType" is required, and can be followed by an optional child "attributeValueChoice", but "attributeValueChoice" cannot appear before "attributeType" in the sequence.

Thales Activation:

- "Hash"'s type of being a base64Binary is properly handled according to the XSD specification's definition of the type.

- "ProtectionKey" allows a user to choose it child to be either "ProtectionKeyInput" or "ProtectionKeyOutput", but not both.
- "execution_count" is a child of "license_properties", and its value is allowed to be either 0 or a positive number.

SEPA:

- "PaymentInstructionInformation3_EPC132-08_SCT_C2B_2019_V1.0_Update" contains exactly 13 children, matching the 13 names mentioned in the XSD.
- "AddtlRmtInf" exists and is not allowed more than once.
- "Cd" is a child of "CreditorReferenceType1Choice_EPC132-08_SCT_C2B_2019_V1.0_Update"

UBL:

- All 15 files are properly imported.
- "ProcedureCode", "PromotionalEventTypeCode" and "QualityControlCode" are root elements (randomly selected a few elements out of the hundreds that are allowed as root, yet are defined in another file). Note that the comments in the project states that the designers do not wish these elements to be root elements, but this contradicts the XSD specification, so they should be treated as root elements.
- Handle "FileName"'s different namespaces properly.

8.3.3.2 results

Whenever a project is correctly processed, its entry is marked with a ✓. If a tool fails to parse or process an XSD project, its entry is marked with a ✗. If a tool is able to properly parse and process an XSD project, but makes minor mistakes in its output, it is marked with a ✓. The following table shows the result of this analysis:

XSD project	Schut parser generator	XSD to C#	XSD to JSON	XSD Validator	JAXB
Purchase Order	✓	✓	✓	✓	✓
IBM Substitution	✓	✓[2]	✓[4]	✓	✓
IBM Testsuite	✓	✗[5]	✓	✗[7]	✓
SGM	✓	✓	✓[4]	✓	✓
Thales Web Service	✓	✓	✓	✓	✓
Thales Activation	✓	✓	✓	✗[8]	✓
SEPA	✓	✓	✓	✓	✓
UBL	✓[1]	✗[6]	✓[4]	✗[3]	✗[3]

Figure 100: The results of the accuracy analysis.

Footnotes:

1. While it produced an output that nearly matched the file, it was not able to properly distinguish between two types that have the same identifier, but a different namespace.
2. Ignored the substitution group and therefore certain elements were completely thrown away.
3. Failed because the tool cannot deal with imports or includes, therefore it is unable to process any multi-file project.

4. Length constraints were not properly transferred.
5. Works in "/c" mode, but fails in "/d". The reasoning is: "Nested table 'category' which inherits its namespace cannot have multiple parent tables in different namespaces.". While within the context of what it is trying to generate this is a valid complaint, it is considered a valid XSD, and therefore it is still marked as not being able to parse it, even though it makes sense that the tool would like the user to change the XSD syntax.
6. Imports are actually supported, and when all files are directly provided it makes an attempt, but still fails due to some namespace/import conflict that it did not seem able to resolve. Another notable point to make is that it does not try to find the files in the file system, but instead requires the user to explicitly provide each file.
7. Cannot find the "report" element. This is likely an issue related to handling namespaces improperly, though the tool does not give enough information in its error message to be sure.
8. Does not accept the Hash as a Base64 value, even though it does follow XSD's specification of Base64.

Furthermore, if we assume that ✗ means a full failure, while ✓ means a half-failure, the following accuracy percentages can be generated based on this data:

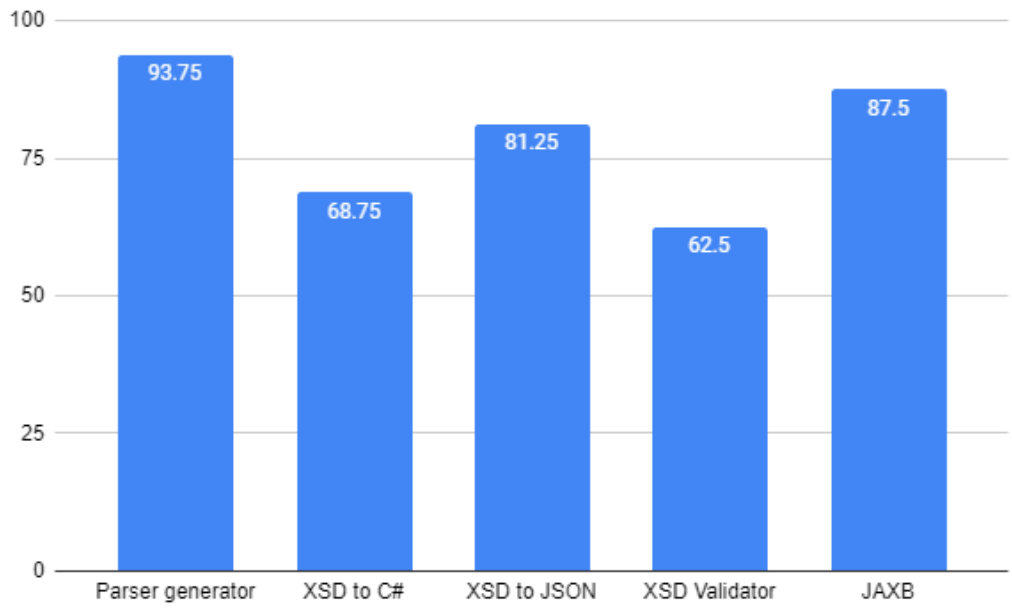

Figure 101: A column chart representing the accuracy of each individual tool.

Firstly, the most crucial fact to conclude from the accuracy results is that the Schut parser generator performs really well, with only one project being problematic due to poor support for namespaces. Namespaces are simply a challenging feature to support with the specific parser type that is being used within this project. DOM parsers such as general-purpose XML parsers are more suited for this purpose. The Schut parser generator outperforms all other projects, with JAXB coming very close and likely having much better support for namespaces, but simply not supporting imports and includes properly.

Notably, for IBM Substitution, every project is able to parse it, but some produce incomplete results. It is a

relatively simple project to parse, however, contains a few "substitutionGroup" statements. This is a relatively niche feature that is infrequently used within XSD. As such, some tools do not truly support this feature, leading to these results of being able to parse it, but the result being incomplete, as the substitutionGroups are not handled.

Another obvious note is the fact that UBL was not properly parsed by a single project. This is to be expected given its immense size and complexity. Three projects all had the same issue: They did not support imports and includes, which is the feature that links the different project files together. As such, they only read the root folder, which references non-existing elements, as they were defined in other files, and subsequently failed to parse. The Schut parser generator and XSD to JSON both properly support imports. However, both were not able to parse it entirely, with the Schut parser generator having issues with namespaces, while XSD to JSON has issues with restrictions. Another interesting note.

The column chart clearly distinguishes between the accuracy of the different tools. The most exciting finding was the low accuracy of XSD to C#, as this tool is part of the .NET libraries and is, therefore, quite a notable tool. JAXB and the parser generator seem to be clearly on top within the dataset used for this specific accuracy analysis.

9 Discussion

9.1 Summary

This section discusses the results of the project, and uses it to draw conclusions and solve the research questions stated at the start of the project. It found the results to be logical, but certainly interesting. It concludes that the results are highly favourable for a recommendation to actually implement this new development methodology when the amount of parsers needing to be generated are high and all parsers share the same data operation. Furthermore, this section discusses possible issues with the results.

9.2 Conclusion

This research aimed to investigate whether building a custom parser generator is an effective alternate strategy compared to building individual parsers when the parser's data operation remains the same across the different formats. The proof of concept shows that it is not only possible, but also doable in a relatively short development period. Within nine weeks, a full program was developed that achieved the stated task, though it did use existing technologies to generate the final parser.

To answer our first research question, it is clear that generating such a parser generator takes a significant but fair amount of weeks. Nine weeks in the case of the proof of concept, but it can take longer or shorter with other data formats, with longer being a fair assessment, as the data operation was relatively simple compared to operations such as importing 3D models. A good estimation for this specific project is that a programmer could generate multiple parsers within the time it took to build this parser generator, but not a large amount. As such, it has become clear that this novel approach is superior when the amount of parsers needed to be generated is significant. However, the older methodology is more effective than the newer one when just a few parsers need to be generated. Exact numbers cannot be attached to this conclusion, as the development time of both a parser generator and the parsers themselves can vary widely between different projects.

To answer our second question, the performance characteristics look good and acceptable. It is important to note that the generation of parsers is a one-time process per parser, and is therefore allowed to take a long time. Despite that, most parser generation tasks take mere seconds to perform. The only true exceptions were UBL, an absolutely enormous project that justifies its long generation time, and Thales Web Service, which took an astonishing amount of time given the project size, clearly showing it is a project with a vast complexity compared to other projects.

Our third question can be answered through the accuracy analysis. The accuracy analysis shows that the tool outshines existing state-of-the-art tools in many applications, thanking its success through support for modern features such as importing and substitution groups. Its accuracy is therefore on a highly acceptable level.

The fourth research question became clear throughout the technical description of the parser generator. Its limitations lie in the strictness of specific grammars like UBNF compared to general-purpose XML parsers. This strictness leads to certain features like arbitrary namespaces being very difficult to support and taking considerable development time to implement. Furthermore, the fact that one still needs to work out a full format description for a parser is not an easy task, and therefore the generation of the parsers id not instant and still requires time to develop a format. Finally, its requirement to build a custom parser generator at the start of any project means a high up-front investment.

The fifth question was answered earlier in the conclusion. Problems that require the generation of many parsers that all have the same data operation can be solved more quickly by this novel approach. There is a threshold for the number of parsers one needs where it goes from the old approach to the new approach being a better workflow.

However, this threshold highly depends on the project and its details and is therefore hard to state.

The last research question was solved throughout section 8.2. It became clear that the limitatoins fall into three categories:

- Missing features within UBNF
- Missing features within the parser builder
- Bugs and missing features within the generated parsers.

The section stated a list of problems and recommended solutions, some of which were implemented during this project. As such, it can be concluded that the technical limitations of the company's toolset mainly are in the category of experimental features that require maturing to improve effectiveness and stability.

As such, the final conclusion that can be drawn, which solves our original main research question, is that this new methodology is an effective development workflow for tasks which require a large number of parsers that all perform the same data operation on different formats. Its development time is of reasonable proportions, while resulting accuracy and performance of such a system would likely be in a highly acceptable range.

9.3 Reflection

Like any research, not every part of the research went flawlessly, and there are some points to consider while interpreting these results.

The first concern is how the design of the parser generator affects the accuracy results. The parser generator is overly generous with accepting the format for certain parts of the XSD. Sometimes an XSD is poorly formatted but still accepted by the tool because it can deduce what was meant. While all tested files are, of course, actually valid, and therefore this deduction is not of relevance, other tools may fail at points that are ignored by the parser generator as it considers it too strict to check for. The tool might have been less accurate if it had been more strict with the format. Therefore, it may be unfair to compare against the other tools, as they try harder to ensure the format is absolutely valid, leading to easier accidental rejection of formats.

A second concern is the number of projects being used to compare. Eight projects are used for performance and accuracy analysis. More projects may have caused more thorough results. However, the counter-argument is that the total size of all eight projects combined is enormous, leading to plenty of testing of different features. The projects were chosen so that all important features within XSD are tested one way or another. To exemplify: The UBL project is larger than all other projects combined, yet it is only considered a single project. As such, many small project may be equivalent in testing capacity to a few large projects. That said, given that the fail/pass is determined on a per-project basis, having fewer large projects leads to fewer fail/pass results, leading to more large-grained results in the percentage of projects being passed by a tool.

10 Future work

10.1 Summary

This section describes the future work that can be performed to expand upon the results produced by this Bachelor's project. It is split into two parts: The first part is about improvements the company can make to the tools created during this project. The second part of the section will discuss the much more classical description of what researchers can do to expand upon the research results into automating parser generation.

10.2 Future work for the company

During the analysis phase of the company's internal tools, a set of flaws were discovered that were all related to either imperfections in existing implementations or a lack of certain features. Resolving these flaws, discussed in section 8.2, is one of the things that the company can still do. While quite a few of them were fixed during this Bachelor's project, some more deep-rooted and complex flaws are yet to be fixed, and resolving such issues would result in a more robust parser generation tool.

Another big ticket for the company to investigate is the implementation of namespaces within XSD. Namespaces are a complex feature within XSD that does not increase the format's expressiveness but makes managing a large XSD project slightly easier. It is currently not supported to its fullest and is essentially ignored by the existing tool. Proper support for such a feature would be beneficial for the tool, as it will improve the accuracy of the parser generator.

A third improvement for the company to undertake is adding support for the remaining constraints. As discussed in section 5.3.6, some constraints were directly implemented, while others need to be handled through post-processing. The eventual goal of the company is to check all constraints during parsing. As such, implementing the remaining constraints is a task that would accomplish this goal. That said, some constraints are somewhat challenging to implement, so this may result in a project that takes significant time to complete.

10.3 Future work for researchers

Research into this topic can be expanded in a multitude of ways. To start, this research was a proof of concept for building an automated parser generator for specific data operations. While certainly demonstrative, it is only a single example of this methodology. It would be highly beneficial for repeated success, and therefore research into parser generators for other data operations would solidify the theory that this is a practical approach. Essentially, it would be beneficial for the research to be repeated on other use cases.

Another interesting investigation, which is somewhat related to the point of repeating the research, is increasing the complexity of the parser generator. Currently, the parser generator builds a parser for importing into a data structure. While certainly no easy task, it does not compare to the more difficult specific data operations, like importing 3D models into rendering tools or automatically generating PDFs from XML files. Implementing parser generators for such data operations may be an even more convincing proof of this methodology's effectiveness.

Further research that would be useful is performance comparison. Currently, the parser generator is compared against other tools based on accuracy, and performance is measured purely in absolute values. It may be helpful to compare the performance of parser generators with other XSD tools. The roadblock that stopped this from happening within this Thesis is that other XSD tools are all in an entirely different performance class compared to parser generators, as other tools perform very simplistic operations such as XSD schema validation or simple code generation. These are not comparable to the parser generator. However, it may be possible to compare the

tool's performance if many variables are taken into account, which would be something for other researchers to investigate.

11 Glossary

- **BNF**: Backus–Naur form. A language used to describe the grammar of a file format, like a programming language or data storage format.
- **CPU:** Central Processing Unit. One of the core components of a computer, responsible for executing instructions.
- **Conflict**: A conflict within this context is when a parser or tokeniser has multiple directions it can travel in a state machine. It may not need to stop the system, but the system will need to make assumptions that are not always correct.
- **Constructor:** A function called by a class that builds the class and sets up its internal data.
- **DRAM:** Main Memory, used to store temporary information.
- **Directed graph**: A data structure that consists out of nodes and edges, where every node is connected through other nodes using edges, and each edge has a direction (so if there is an edge between node A and node B, one of them is the start node and the other is the end node).
- **Encoding**: How a character within a piece of text is represented in binary. Nowadays the amounts of bytes used for a character is often variable.
- **GPU:** Graphics Processing Unit. Similar to the CPU, but specialised to be really good at executing specific instructions.
- **GUI**: Graphical User Interface. An interface that is rendered in the front of the user, and the user can interact with through their mouse and keyboard. It is the main bridge between the user and the underlying code, where the user instructs the program to do something through the GUI.
- **General-purpose**: A system that is general-purpose can deal with a lot of different cases. It is the opposite of a specialised system, and can therefore deal with a large range of applications, though it often trades off in efficiency.
- **Grammar**: In the context of this thesis, a grammar is a definition of how a structured file should be formatted. It specifies exactly which content is valid, and which is not.
- **IDE**: Integrated Development Environment. A software package that contains all tools necessary for a programmer to write software.
- **IT**: Information Technology. A term used to refer to any field or application that involves maintaining or developing digital systems.
- **Kleene star:** An operator within regular expressions that allows one to state that an element should be repeated between 0 and an infinite amount of times. The character used for it is "*".
- **Metadata**: Data that is not part of the actually used content, but necessary to store information about the data being used. In this research, it refers to the metadata added to a grammar to specify information such as the starting rule.
- **Parser generator**: A piece of software that can automatically build parsers using a specification of the format it is supposed to parse
- **Parser**: A piece of software that is able to read a string, split it up into its primitives, confirm that it follows a certain format, and then process the primitives contained within the string.

- **Pipeline**: A sequence of steps that are executed from first to last to turn a certain input into a certain output.
- **Production rule**: The core component of a grammar. It specifies the structure of a file by specifying a sequence of elements that should appear in a row in the file. Such sequences can consist out of characters and other production rules, allowing for a recursive description of a file structure.
- **Reactive code**: Code that reacts to events being called by the parser, like a rule being matched or a specific token being encountered.
- **Schut array**: A data structure developed by Schut B.V. Internally simply referred to as "Array". It is essentially a heterogeneous array, where each element can be of different types. Interestingly, a Schut array is also allowed to be a type, leading to a hierarchical structure of nested Schut arrays.
- **State machine**: A directed graph that can be used to perform state transitions, which are a way to parse files and support regular expressions.
- **Storage:** Secondary Memory, used to store more permanent information.
- **Token contamination**: A term developed during the paper, which refers to the idea of the parser taking a longer token even though it is of no relevance to the current grammar rule being investigated.
- **Token**: A group of characters within a piece of text that form a single element within the text's structure. The characters that make up a word are often grouped together within a token.
- **UBNF**: An extension of BNF developed by Schut B.V. It mainly focuses on two aspects: Adding syntactic sugar to make grammars significantly shorter and easier to read, and adding support for directly using UBNF grammars to import into Schut arrays.
- **W3C**: World Wide Web Consortium. An organisation tasked with developing standards used within the web [37].
- **XML**: eXtended Markup Language. A markup language that is used to store data in a format that is readable by both humans and computers.
- **XSD**: A standard developed by W3C to define XML formats. It allows a user to impose constraints on an XML file, including what tags should appear, which tags can be children of which other tags, and other constraints like the amount of times a tag is allowed to appear.

12 References

[1] R. Fernandes and M. Raghavachari. Inflatable xml processing. *MIDDLEWARE 2005, PROCEEDINGS*, 3790:144–163, 2005. 6th International Middleware Conference, Grenoble, FRANCE, NOV 28-DEC 02, 2005.

[2] David Yen, Shi Huang, and Cheng-Yuan Ku. The impact and implementation of xml on business-to-business commerce. *Computer Standards & Interfaces*, 24:347–362, Sep 2002.

[3] Sybase. *Introduction to Sybase Message Bridge for Java*, volume 1, page 3–3. Sybase, 5.2 edition, Jan 2005.

[4] W3C. *Extensible Markup Language (XML) 1.0*, 5th edition, Nov 1998.

[5] W3C. *XML Schema Part 0: Primer*, 2nd edition, Oct 2004.

[6] Michael Scognamiglio. An introduction to regular expressions., Oct 2020. https://towardsdatascience.com/an-introduction-to-regular-expressions-5dd762afc5e4.

[7] Taylor L Booth. Sequential machines and automata theory. *New York: Wiley*, 1967.

[8] W.H. Hesselink, J.A. Pérez, and J.J. van de Gronde. Languages & machines. *Languages & Machines course, RUG*, Mar 2022.

[9] Boštjan Slivnik. LLLR Parsing: a Combination of LL and LR Parsing. In Marjan Mernik, José Paulo Leal, and Hugo Gonçalo Oliveira, editors, *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] Xiaoying Wu and Dimitri Theodoratos. A survey on XML streaming evaluation techniques. *VLDB JOURNAL*, 22(2):177–202, APR 2013.

[11] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 19(10):1381–1403, OCT 2007.

[12] Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami. 2LP: A double-lazy XML parser. *INFORMATION SYSTEMS*, 34(1):145–163, MAR 2009.

[13] E Perkins, M Matsa, MG Kostoulas, A Heifets, and N Mendelsohn. Generation of efficient parsers through direct compilation of XML Schema grammars. *IBM SYSTEMS JOURNAL*, 45(2):225–244, 2006.

[14] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to xml parsing. *2006 7TH IEEE/ACM INTERNATIONAL CONFERENCE ON GRID COMPUTING*, pages 223–230, SEP 2006.

[15] Yi-jun Bei, Gang Chen, Jin-xiang Dong, and Ke Chen. Bottom-up mining of XML query patterns to improve XML querying. *JOURNAL OF ZHEJIANG UNIVERSITY-SCIENCE A*, 9(6):744–757, JUN 2008.

[16] Hanyong Choi and Sungho Sim. A Study on Efficiency of Markup Language Using DOM Tree. *WIRELESS PERSONAL COMMUNICATIONS*, 86(1, SI):143–163, JAN 2016.

[17] Sang-Kyun Kim, Myungcheol Lee, and Kyu-Chul Lee. Validation of xml document updates based on xml schema in xml databases. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 98–108, SEP 2003.

[18] A Balmin, Y Papakonstantinou, and V Vianu. Incremental validation of XML documents. *ACM TRANSACTIONS ON DATABASE SYSTEMS*, 29(4):710–751, DEC 2004.

[19] Joe Tekli, Richard Chbeir, Agma J. M. Traina, Caetano Traina, Jr., and Renato Fileto. Approximate XML structure validation based on document-grammar tree similarity. *INFORMATION SCIENCES*, 295:258–302, FEB 20 2015.

[20] Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, Daniel Rodriguez-Cerezo, and Jose-Luis Sierra. Building XML-Driven Application Generators with Compiler Construction Tools. *COMPUTER SCIENCE AND INFORMATION SYSTEMS*, 9(2):485–504, JUN 2012.

[21] Wei Zhang and Robert A. van Engelen. Tdx: A high-performance table-driven xml parser. *PROCEEDINGS OF THE 44TH ANNUAL SOUTHEAST REGIONAL CONFERENCE*, page 726–731, MAR 2006.

[22] Daniel Hoffman, Hong-Yi Wang, Mitch Chang, David Ly-Gagnon, Lewis Sobotkiewicz, and Paul Strooper. Two case studies in grammar-based test generation. *JOURNAL OF SYSTEMS AND SOFTWARE*, 83(12, SI):2369–2378, DEC 2010.

[23] Johannes Griss, Florian Reisinger, Henning Hermjakob, and Juan Antonio Vizcaino. jmzReader: A Java parser library to process and visualize multiple text and XML-based mass spectrometry data formats. *PROTEOMICS*, 12(6):795–798, MAR 2012.

[24] WD MAURER. Semantic extension of BNF. *INTERNATIONAL JOURNAL OF COMPUTER MATHEMATICS*, 3(2-3):157–176, 1972.

[25] Hong Zhu. An institution theory of formal meta-modelling in graphically extended BNF. *FRONTIERS OF COMPUTER SCIENCE*, 6(1):40–56, FEB 2012.

[26] Paul B. Mann. A translational BNF grammar notation (TBNF). *ACM SIGPLAN NOTICES*, 41(4):16–23, APR 2006.

[27] Hong Zhu. On the theoretical foundation of meta-modelling in graphically extended bnf and first order logic. *2010 4TH IEEE INTERNATIONAL SYMPOSIUM ON THEORETICAL ASPECTS OF SOFTWARE ENGINEERING*, pages 95–104, 2010.

[28] Roman R. Redziejowski. More About Converting BNF to PEG. *FUNDAMENTA INFORMATICAE*, 133(2-3):257–270, JAN 2014.

[29] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *IFIP CONGRESS*, pages 1–20, JUN 1959.

[30] Joel E. Denny and Brian A. Malloy. The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *SCIENCE OF COMPUTER PROGRAMMING*, 75(11, SI):943–979, NOV 1 2010.

[31] Nobuyuki Kobayashi, Hiromitsu Shiina, and Sigeru Masuyama. Transformation from an unrestricted LR(k) grammar into an unrestricted LR(1) grammar. *INFORMATION-AN INTERNATIONAL INTERDISCIPLINARY JOURNAL*, 11(2):215–227, MAR 2008.

[32] Joel E. Denny and Brian A. Malloy. The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *SCIENCE OF COMPUTER PROGRAMMING*, 75(11, SI):943–979, NOV 1 2010.

[33] J Fong, A Fong, HK Wong, and P Yu. Translating relational schema with constraints into XML schema. *INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 16(2):201–243, APR 2006.

[34] J Fong and SK Cheung. Translating relational schema into XML schema definition with data semantic preservation and XSD graph. *INFORMATION AND SOFTWARE TECHNOLOGY*, 47(7):437–462, MAY 15 2005.

[35] Benbin Chen, Chung-Ta King, Xiaochao Li, and Donghui Guo. A high quality compiler tool for application-specific instruction-set processors with library and parallel supports. *MULTIMEDIA TOOLS AND APPLICATIONS*, 76(4):5905–5926, FEB 2017.

[36] IBM. Omnibus specification. https://www.ibm.com/docs/en/netcoolomnibus/8.1?topic=library-minimal-non-greedy-quantifiers.

[37] W3C. World wide web consortium information. https://www.w3.org/Consortium/.

13 Changelog

Date	Sections	Changes	Version
21-03-2022	Titlepage & Changelog	Set up title page and changelog	1.0.0
07-04-2022	Everywhere	Set up an initial structure of sections	1.1.0
12-04-2022	Everywhere	Added multiple new sections to the skeleton, and started refining subsections	1.1.1
25-04-2022	Everywhere	Refined multiple sections, added some basic elements to the sections, started writing the introduction	1.2.0
26-04-2022	Introduction	Continued writing and refining the introduction, added the automation diagram	1.2.1
26-04-2022	Glossary	Added lots of terminology to the glossary	1.2.2
27-04-2022	Introduction	Finished the first draft of the introduction	1.2.3
27-04-2022	Background	Finished the first draft of the XML background	1.2.4
27-04-2022	Architecture	Started writing the pipeline section, made some initial diagram drafts.	1.2.5
28-04-2022	Architecture & Glossary	Finished the pipeline section's first draft and worked on the terminology.	1.2.6
29-04-2022	Background	Finished the first draft of the XSD section.	1.2.7
29-04-2022	Background	Finished the first draft of the State Machine section.	1.2.8
29-04-2022	Appendix	Set up some more appendix content and fixed multiple page-break issues.	1.2.9
30-04-2022	Architecture	Wrote the first half of the manual subsection.	1.2.10
01-05-2022	Architecture	Significant expanded the manual subsection.	1.2.11
02-05-2022	Everywhere	Redesigned multiple sections & finished the first draft of the manual section.	1.3.0
03-05-2022	Discussion	Wrote random order and token contamination.	1.3.1
03-05-2022	XSD parsing	Started with XSD parsing.	1.3.2
03-05-2022	Everywhere	Updated the syntax highlighting for XML, C and UBNF to be a lot more visually pleasing.	1.3.3
04-05-2022	Everywhere	Improved syntax highlighting of UBNF.	1.3.4
04-05-2022	XSD parsing	Wrote the Output section.	1.3.5
04-05-2022	Glossary	Updated the terminology.	1.3.6
04-05-2022	XSD parsing	Got started with the XSD hierarchy section.	1.3.7
05-05-2022	Background	Finished the Regular Expressions section.	1.3.8
06-05-2022	Background	Finished the parser section.	1.3.9
06-05-2022	XSD parsing	Finished the XSD hierarchy section.	1.4.0
06-05-2022	Introduction	Added document layout and acknowledgements.	1.4.1
07-05-2022	Background	Wrote the Context-free grammar and BNF sections.	1.4.2
09-05-2022	Changelog	Improved formatting and styling of the changelog.	1.4.3
09-05-2022	Introduction	Rewrote the introduction to better include my second research question.	1.5.0
09-05-2022	XSD Parsing	Wrote the first draft of XSD defined in UBNF.	1.5.1
10-05-2022	Architecture	Wrote the automated section.	1.5.2

Date	Sections	Changes	Version
10-05-2022	Appendix	Added appendix B and C.	1.5.3
10-05-2022	Everywhere	Restructured the document.	1.6.0
10-05-2022	Background	Wrote the Schut array section.	1.6.1
11-05-2022	Everywhere	Style improvements.	1.6.2
11-05-2022	XSD conversion	Started writing the XSD conversion section.	1.6.3
12-05-2022	Background	Wrote the first half of the UBNF section.	1.6.4
13-05-2022	Background	Wrote the second half of the UBNF section.	1.6.5
13-05-2022	Analysis and Parser building	Restructured both sections as they are closely linked together.	1.7.0
13-05-2022	Analysis	Wrote the encodings section.	1.7.1
13-05-2022	Glossary	Updated the glossary	1.7.2
13-05-2022	XSD Conversion	Wrote the tag replacement section	1.7.3
13-05-2022	Front page	Added new content to the front page	1.7.4
14-05-2022	Analysis	Set up the structure for the analysis section	1.7.5
14-05-2022	Everywhere	Lots of small changes in the structure	1.7.6
16-05-2022	Discussion & Analysis	Moved a few sections around	1.7.7
16-05-2022	Parser Building	Wrote the tokenisation section	1.7.8
16-05-2022	Discussion	Set up the discussion structure	1.8.0
16-05-2022	Introduction	Updated the research questions	1.8.1
16-05-2022	Analysis	Started writing the empty production rules section	1.8.2
16-05-2022	Parser Building	Started writing the error handling section	1.8.3
17-05-2022	Parser Building	Finished the error handling section	1.8.4
17-05-2022	Parser Building	Updated the tokenisation section	1.8.5
17-05-2022	Analysis	Set up a system to mark as resolved/unresolved	1.8.6
19-05-2022	Analysis	Wrote the non-greedy operator section	1.8.7
19-05-2022	XSD Parsing	Wrote the pre-processor section	1.8.8
22-05-2022	Related work	Wrote the first draft of the related work section	1.9.0
22-05-2022	XSD parsing	Many small changes and extensions	1.9.1
22-05-2022	Analysis	Wrote the bounded repetition section	1.9.2
24-05-2022	Everywhere	Lots of grammatical improvements following proofreading	1.9.3
24-05-2022	XSD conversion	Wrote the post-processor section	1.9.4
26-05-2022	Introduction	Rewrote the introduction based on feedback	1.10.0
26-05-2022	Analysis	Wrote the error handler section	1.10.1
26-05-2022	XSD Parsing	Added links to the state machines for the XSD parser	1.10.2
27-05-2022	Analysis	Set up a framework table for the performance reporting	1.10.3
27-05-2022	Everywhere	Various small improvements based on feedback	1.10.4
27-05-2022	Everywhere	Improved multiple summaries	1.10.5
28-05-2022	Parser building	Rewrote multiple sections	1.10.6
29-05-2022	Parser building & XSD parsing	Restructured both sections and wrote multiple sections	1.11.0
30-05-2022	Parser building	Started writing about the state machines	1.11.1
31-05-2022	Parser building	Wrote encodings section	1.11.2

Date	Sections	Changes	Version
31-05-2022	Analysis	Wrote the control characters section	1.11.3
31-05-2022	References	Added more sources and citations throughout the paper	1.11.4
31-05-2022	Analysis	Wrote the static array additions section	1.11.5
31-05-2022	Introduction	Wrote first draft of scientific integrity	1.11.6
01-06-2022	Introduction	Finished scientific integrity	1.11.7
01-06-2022	Architecture	Rewrote the pipeline section to be much more readable	1.11.8
02-06-2022	Introduction	Wrote the acknowledgements	1.11.9
02-06-2022	Architecture	Removed the pipeline creation section	1.11.10
02-06-2022	Architecture	Wrote the technology stack	1.11.11
02-06-2022	Analysis	Reworked the setup for performance and accuracy testing	1.12.0
03-06-2022	Everywhere	A few hundred text flow and grammatical improvements	1.12.1
03-06-2022	Everywhere	Shortened text and removed unnecessary content	1.12.2
03-06-2022	Introduction	Rewrote some parts	1.12.3
04-06-2022	Analysis	Wrote the analysed projects	1.12.4
04-06-2022	Analysis	Wrote the summary	1.12.5
06-06-2022	Parser builder	Wrote terminal state machine	1.12.6
06-06-2022	Parser builder	Wrote grammar state machine	1.12.7
07-06-2022	Analysis	Wrote XSD projects	1.12.8
07-06-2022	Analysis	Wrote XSD tools	1.12.9
07-06-2022	Analysis	Wrote Hardware	1.12.10
07-06-2022	Analysis	Wrote performance section, results are yet to be filled in	1.12.11
09-06-2022	Analysis	Filled in the results of performance testing	1.12.12
10-06-2022	Everywhere	Minor improvements based on feedback.	1.12.13
10-06-2022	Analysis	Filled in the first half of the accuracy results.	1.12.14
10-06-2022	XSD conversion	Added the class diagram.	1.12.15
11-06-2022	Analysis	Filled in the rest of the accuracy results.	1.12.16
13-06-2022	Analysis	Reworked the performance results and added appendix D to take an average of 5 test runs.	1.13.0
13-06-2022	Related Work	Fixed a few issues within this section based on feedback.	1.13.1
13-06-2022	Title page	Wrote the abstract.	1.13.2
14-06-2022	Analysis	Finished the new average performance results.	1.13.3
14-06-2022	Parser Building	Wrote the Static Content Section.	1.13.4
14-06-2022	Analysis	Finished the conditions of accuracy analysis and repeated testing.	1.13.5
15-06-2022	Everywhere	Various grammar and text-flow improvements.	1.13.6
15-06-2022	Analysis & Parser Building	Wrote the 2 sections related to the grammar parameters.	1.13.7
15-06-2022	Discussion	Wrote the conclusion.	1.13.8
15-06-2022	Discussion	Wrote the reflection.	1.13.9
16-06-2022	Discussion	Grammar improvements and wrote summary.	1.13.10
16-06-2022	XSD conversion	Wrote constraints.	1.13.11
17-06-2022	Discussion	Wrote "The results".	1.13.12

Date	Sections	Changes	Version
17-06-2022	Analysis	Improved empty production rules.	1.13.13
17-06-2022	Analysis	Expanded the footnotes of accuracy analysis to leave a note for every failure.	1.13.14
20-06-2022	Future work	Wrote future work for the company.	1.13.15
20-06-2022	Future work	Wrote future work for researchers.	1.13.16
20-06-2022	Analysis	Added results for parser performance.	1.13.17
20-06-2022	Discussion	Added a discussion on the results of the parser performance.	1.13.18
22-06-2022	Everywhere	Grammar and text flow improvements.	1.13.19
22-06-2022	Analysis	Moved the interpretation of the results from the conclusion to the analysis.	1.13.20
23-06-2022	Everywhere	Updated all resources.	1.13.21
23-06-2022	Appendix	Updated grammars to fit within page limit.	1.13.22
23-06-2022	XSD conversion	Updated the constraints section.	1.13.23
23-06-2022	Everywhere	Many grammar improvements.	1.13.24
23-06-2022	Parser building	Wrote parameter extension.	1.13.25
24-06-2022	XSD conversion	Wrote substitution.	1.13.26
24-06-2022	Everywhere	Text flow improvements.	1.13.27
24-06-2022	Future work	Expanded the section with new suggestions.	1.13.28
06-07-2022	Glossary	Sorted entries alphabetically.	1.13.29
06-07-2022	References	Fixed an improper reference.	1.13.30

A XSD grammar

```
//Metadata
( Start Rule ) := <XSD> ;

( Single Line Comment ) := "//" ;

( Multi Line Comment ) := "<!--" "-->" ;

( Keep Comments ) := false ;
//End metadata

//Terminals
{NameCh} := {%AlphaNumeric} + [_-.];

{URLChrs} := {%AlphaNumeric} + [ :._-/#];

{PatternCh} := {%AlphaNumeric} + [*+%_-?!@#$=\\.,;^$&|\[\]{}() ];

{ContentCh} := {NameCh} + [()];

{IntegerCh} := {%Number} + [-.+e];

{NamespaceCh} := {%AlphaNumeric} + [#-:./_];

<$Name> := "\"" {NameCh}+ "\"";

<$VersionString> := "\"" {%Number}+ "." {%Number}+ "\"";

<$DefaultFixedString> := "\"" {%AlphaNumeric}+ "\"";

<$URL> := "\"" {URLChrs}+ "\"";

<$Integer> := "\"" {IntegerCh}+ "\"";

<$Regex> := "\"" {PatternCh}+ "\"";

<$Content> := {ContentCh}+;

<$Namespace> := "\"" {NamespaceCh}+ "\"";

//End terminals

//Non-terminals
//Different users of the XSD format switch between "xs" and "xsd" as namespace, so support both
<Namespace; RemoveFromResult=true> := "xs:" | "xsd:";

//Root element
<XSD; Alias=true> := <XML_Version> <Schema>;

<XML_Version; ArrayId = "Metadata"> := ~"<?xml" <Version_String> <Encoding_String>? ~"?>";
<Version_String; ValueRef = "VersionString"; ArrayId = "Version"> := "version=" <$VersionString>;
<Encoding_String; ValueRef = "Name"; ArrayId = "Encoding"> := "encoding=" <$Name>;
```

```
<Schema> := <SchemaStartOpen> <SchemaHeader> ~”>” <Annotation>? <Root>* <SchemaEnd>;

<SchemaStartOpen; RemoveFromResult=true> := ~”<” <Namespace>? ”schema”;

//Schema metadata logic
<SchemaHeader> := <SchemaParameter>*;

<SchemaParameter; Alias=true>:= <XMLSchemaURL> | <XMLSchemaURLMisc> | <targetNamespace> |
<elementFormDefault> | <attributeFormDefault> | <XMLSchemaVersion>;

<XMLSchemaURL; ValueRef=”URL”> := ~”xmlns=” <$URL>;
<XMLSchemaURLMisc; ValueRef=”URL”> := ~”xmlns:” <$Content> ~”=” <$URL>;
<targetNamespace; ValueRef=”URL”> := ~”targetNamespace=” <$URL>;
<elementFormDefault; Type = WString> := ~”elementFormDefault=” <FormValues>;
<attributeFormDefault; Type = WString> := ~”attributeFormDefault=” <FormValues>;
<XMLSchemaVersion; ValueRef=”VersionString”> := ~”version=” <$VersionString>;

<FormValues; Alias=true> := (”\”qualified\”” | ”\”unqualified\””);
//End schema metadata logic

//Annotation logic
//A basic implementation of annotations are supported,
//however some usages of the annotations section are impossible to support in UBNF
//As such, annotations are removed in the pre−processor.
<Annotation; RemoveFromResult=true> :=
~”<” <Namespace>? ~”annotation>”
<AnnotationAppinfo>? <AnnotationDocumentation>?
~”</” <Namespace>? ~”annotation>”;

<AnnotationAppinfo> :=
~”<” <Namespace>? ~”appinfo>”
<$Content>+
~”</” <Namespace>? ~”appinfo>”;

<AnnotationDocumentation> :=
~”<” <Namespace>? ~”documentation>”
<$Content>+
~”</” <Namespace>? ~”documentation>”;
//End annotation logic

//Import logic
<Import> :=
//Officially Schema location is optional, but an import statement without it cannot be processed.
~”<” <Namespace>? ~”import” (<ParameterNamespace> <ParameterSchemaLocation>)! ~”/>” |
~”<” <Namespace>? ~”include” <ParameterSchemaLocation> ~”/>”;

//End import logic

<SchemaEnd; RemoveFromResult=true> := ~”</” <Namespace>? ~”schema>”;

//Tags that can be at the root of the XSD (Note that there can be multiple roots).
<Root; Alias = true> :=
<Element> | <ComplexType> | <SimpleType> | <AttributeGroup> |
```

```
<Import> | <Group> | <Attribute>;

//Element type logic
<Type; Alias = true> := <ComplexType> | <SimpleType>;

//Complex type logic
<ComplexType> :=
~"<" <Namespace>? ~"complexType" ((<ParameterName> <ParameterId> <ParameterMixed> <ParameterAbstract>)!)? ~">"
<Annotation>? (<SimpleContent> | <ComplexContent> | ((<Group>|<All>|
<Choice>|<Sequence>)? ((<Attribute>|<AttributeGroup>)* <AnyAttribute>?)))
~"</" <Namespace>? ~"complexType" ~">";

<ComplexContent> :=
~"<" <Namespace>? ~"complexContent" <ParameterId>? ~">"
<Annotation>? (<Extension> | <Restriction>)
~"</" <Namespace>? ~"complexContent" ~">";

<SimpleContent> :=
~"<" <Namespace>? ~"simpleContent" <ParameterId>? ~">"
<Annotation>? (<Extension> | <Restriction>)
~"</" <Namespace>? ~"simpleContent" ~">";

<SimpleType> :=
~"<" <Namespace>? ~"simpleType" ((<ParameterName> <ParameterId>)!)? ~">"
<Annotation>? (<Restriction> | <List> | <Union>)
~"</" <Namespace>? ~"simpleType" ~">";
//Split the > off to a seperate token, as the token will otherwise match at the start tag instead of "simpleType" and fail.
//End Complex type logic

//Extension logic
<Extension> := <ComplexExtension> | <SimpleExtension>;

<ComplexExtension; Alias=true> :=
~"<" <Namespace>? ~"extension" <ParameterId>? <BaseParameter> ~">"
<Annotation>? ((<Group> | <All> | <Choice> | <Sequence>)?
((<Attribute>|<AttributeGroup>)* <AnyAttribute>?))
~"</" <Namespace>? ~"extension" ~">";

<SimpleExtension; Alias=true> := ~"<" <Namespace>? ~"extension" <ParameterId>? <BaseParameter> ~"/>";
//End extension logic

//AttributeGroup logic
<AttributeGroup> := <AttributeGroupComplex> | <AttributeGroupSimple>;
<AttributeGroupComplex; Alias=true> :=
~"<" <Namespace>? ~"attributeGroup" <ParameterName>? <ParameterRef>? ~">"
<Annotation>? (<Attribute>|<AttributeGroup>)* <AnyAttribute>?
~"</" <Namespace>? ~"attributeGroup" ~">";
<AttributeGroupSimple; Alias=true> := ~"<" <Namespace>? ~"attributeGroup" <ParameterName>? <ParameterRef>? ~"/>";
//End attributeGroup logic

//Value aggregation logic
<List> := <ListComplex> | <ListSimple>;
<ListComplex; Alias=true> :=
~"<" <Namespace>? ~"list" <ParameterItemType> ~">"
```

```
<Annotation>? <SimpleType>?
~"</" <Namespace>? ~"list>";
<ListSimple; Alias=true> := ~"<" <Namespace>? ~"list" <ParameterItemType> ~"/>";

<Union> := <UnionComplex> | <UnionSimple>;
<UnionComplex; Alias=true> :=
~"<" <Namespace>? ~"union" <ParameterMemberType>? ~">"
<Annotation>? <SimpleType>*
~"</" <Namespace>? ~"union" ~">";
<UnionSimple; Alias=true> := ~"<" <Namespace>? ~"union" <ParameterMemberType> ~"/>";


//End value aggegration logic

//Scope logic
<Key; Alias=true> :=
~"<" <Namespace>? ~"key" (<ParameterId> <ParameterName>)! ~">"
<Annotation>? <Selector> <Field>+
~"</" <Namespace>? ~"key" ~">";

<KeyRef; Alias=true> :=
~"<" <Namespace>? ~"keyref" (<ParameterId> <ParameterName>)! ~">"
<Annotation>? <Selector> <Field>+
~"</" <Namespace>? ~"keyref" ~">";

<Unique; Alias=true> :=
~"<" <Namespace>? ~"unique" (<ParameterId> <ParameterName>)! ~">"
<Annotation>? <Selector> <Field>+
~"</" <Namespace>? ~"unique" ~">";

//End scope logic

//Indicators handle how the list of children are treated. Either a user has to choose a child to use,
//can use them all, or has to use them in a certain sequence.
<Indicator; Alias = true> := <Sequence> | <All> | <Choice>;

<Sequence> :=
~"<" <Namespace>? ~"sequence" ((<ParameterId> <ParameterMaxOccurs> <ParameterMinOccurs>)!)? ~">"
<Annotation>? (<Element> | <Group> | <Choice> | <Sequence> | <Any>)+
~"</" <Namespace>? ~"sequence" ~">";

<All> :=
~"<" <Namespace>? ~"all" ((<ParameterId> <ParameterMaxOccurs> <ParameterMinOccurs>)!)? ~">"
(<Element> | <BasicAttribute>)+
~"</" <Namespace>? ~"all" ~">";

<Choice> :=
~"<" <Namespace>? ~"choice" ((<ParameterId> <ParameterMaxOccurs> <ParameterMinOccurs>)!)? ~">"
<Annotation>? (<Element> | <Group> | <Choice> | <Sequence> | <Any>)+
~"</" <Namespace>? ~"choice" ~">";
//Split the > off to a seperate token, as the token will otherwise match at the start tag instead of "choice" and fail.
//End element type logic

//Selector logic
```

```
<Selector> := <ComplexSelector> | <SimpleSelector>;

<ComplexSelector; Alias=true> :=
~"<" <Namespace>? ~"selector" <ParameterId>? <ParameterXPath> ">"
<Annotation>?
~"</" <Namespace>? ~"selector>";

<SimpleSelector; Alias=true> := ~"<" <Namespace>? ~"selector" <ParameterId>? <ParameterXPath> ~"/>";
//End selector logic

//Field logic
<Field> := <ComplexField> | <SimpleField>;

<ComplexField; Alias=true> :=
~"<" <Namespace>? ~"field" <ParameterId>? <ParameterXPath> ">"
<Annotation>?
~"</" <Namespace>? ~"field>";

<SimpleField; Alias=true> := ~"<" <Namespace>? ~"field" <ParameterId>? <ParameterXPath> ~"/>";
//End field logic

//Group logic
<Group> := <ComplexGroup> | <SimpleGroup>;
<ComplexGroup; Alias=true> :=
~"<" <Namespace>? ~"group" <ParameterName>? <ParameterRef>? ~">"
<Annotation>? (<All> | <Choice> | <Sequence>)?
~"</" <Namespace>? ~"group" ~">";
<SimpleGroup; Alias=true> := ~"<" <Namespace>? ~"group" <ParameterName>? <ParameterRef>? ~"/>";
//End group logic

//Any logic
<Any> := <ComplexAny> | <SimpleAny>;

<SimpleAny; Alias=true> :=
~"<" <Namespace>? ~"any"
(<ParameterMinOccurs> <ParameterMaxOccurs> <ParameterNamespace>
<ParameterId> <ParameterProcessContents>)!
~"/>";

<ComplexAny; Alias=true> :=
~"<" <Namespace>? ~"any"
(<ParameterMinOccurs> <ParameterMaxOccurs> <ParameterNamespace>
<ParameterId> <ParameterProcessContents>)! ~">"
<Annotation>?
~"</" <Namespace>? ~"any>";
//End any logic

//Restriction logic
<Restriction; Alias=true> := <ComplexRestriction> | <SimpleRestriction>;

//BaseParameter is mandatory for Simple, as it otherwise would not define any restriction.
<SimpleRestriction; ArrayId="Restriction"> := ~"<" <Namespace>? ~"restriction" <BaseParameter> ~"/>";

<ComplexRestriction; ArrayId="Restriction"> :=
```

```
<RestrictionOpen>
  <Annotation>? <RestrictionContent>*
<RestrictionClose>;

<RestrictionOpen; Alias=true> := ~"<" <Namespace>? ~"restriction" <BaseParameter>? ~">";

<RestrictionClose ; RemoveFromResult=true> := ~"</" <Namespace>? ~"restriction>";

<RestrictionContent; Alias=true> :=
<MinLength> | <MaxLength> | <MinInclusive> | <MaxInclusive> | <MinExclusive> |
<MaxExclusive> | <Length> | <TotalDigits> | <FractionDigits> | <WhiteSpace> |
<Pattern> | <Enum> | <Attribute> | <AttributeGroup> | <SimpleType> | <Group> |
<Choice> | <All> | <Sequence>;

///Different types of restrictions
<MinLength ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"minLength value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"minLength value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"minLength" ~">";
<MaxLength ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"maxLength value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"maxLength value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"maxLength" ~">";
<MinInclusive ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"minInclusive value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"minInclusive value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"minInclusive" ~">";
<MaxInclusive ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"maxInclusive value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"maxInclusive value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"maxInclusive" ~">";
<MinExclusive ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"minExclusive value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"minExclusive value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"minExclusive" ~">";
<MaxExclusive ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"maxExclusive value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"maxExclusive value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"maxExclusive" ~">";
<Length ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"length value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"length value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"length" ~">";
<TotalDigits ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"totalDigits value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"totalDigits value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"totalDigits" ~">";
<FractionDigits ; ValueRef="Integer"> :=
~"<" <Namespace>? ~"fractionDigits value" ~"=" <$Integer> ~"/>" |
~"<" <Namespace>? ~"fractionDigits value" ~"=" <$Integer> ~">"
~"</" <Namespace>? ~"fractionDigits" ~">";
<Pattern ; ValueRef="Regex"> :=
~"<" <Namespace>? ~"pattern value" ~"=" <$Regex> ~"/>" |
~"<" <Namespace>? ~"pattern value" ~"=" <$Regex> ~">"
```

```
~"</" <Namespace>? ~"pattern" ~">";
<Enum ; ValueRef="Regex"> :=
~"<" <Namespace>? ~"enumeration value" ~"=" <$Regex> ~"/>" |
~"<" <Namespace>? ~"enumeration value" ~"=" <$Regex> ~">"
~"</" <Namespace>? ~"enumeration" ~">";
<WhiteSpace ; ValueRef="WhiteSpaceValues"> :=
~"<" <Namespace>? ~"whiteSpace value" ~"=" <WhiteSpaceValues> ~"/>" |
~"<" <Namespace>? ~"whiteSpace value" ~"=" <WhiteSpaceValues> ~">"
~"</" <Namespace>? ~"whiteSpace" ~">";

<WhiteSpaceValues> := "preserve" | "replace" | "collapse";
//End restriction logic

//Element logic
//EndTag is necessary because BasicElement breaks when using the ~ operator at the end, due to a UBNF limitation.
<Element> := <ComplexElement> | <BasicElement>;

<ComplexElement; Alias=true> :=
~"<" <Namespace>? ~"element" <ElementParameter>? ~">"
<Annotation>? <Type>? (<Unique> | <Key> | <KeyRef>)*
~"</" <Namespace>? ~"element>"; //Last token is still safe as we have to provide a name at the start of the tag.

<BasicElement; Alias=true> :=
~"<" <Namespace>? ~"element" <ElementParameter>? ~"/>";

//Attribute logic
<Attribute; Alias=true> := <ComplexAttribute> | <BasicAttribute>;

<BasicAttribute; ArrayId = "Attribute"> :=
~"<" <Namespace>? ~"attribute" <AttributeParameter>? ~"/>";

<ComplexAttribute; ArrayId = "Attribute"> :=
~"<" <Namespace>? ~"attribute" <AttributeParameter>? ~">"
<Annotation>? <SimpleType>?
~"</" <Namespace>? ~"attribute>"; //Last token is still safe as we have to provide a name at the start of the tag.

<AnyAttribute> := ~"<" <Namespace>? ~"anyAttribute" ((<ParameterId> <ParameterNamespace>)!)? ~"/>";
//End attribute logic

//Default and fixed parameter types cannot be both specified for an element.
<ElementParameter; Alias = true> :=
(<ParameterId> | <ParameterName> | <ParameterRef> |
<ParameterType> | <ParameterSubstitutionGroup> |
<ParameterDefault> | <ParameterFixed> | <ParameterForm> |
<ParameterMaxOccurs> | <ParameterMinOccurs> |
<ParameterNillable> | <ParameterAbstract> |
<ParameterBlock> | <ParameterFinal>)*;

<AttributeParameter; Alias = true> :=
(<ParameterName> | <ParameterType> | <ParameterUse> | <ParameterFixed> |
<ParameterDefault> | <ParameterForm> | <ParameterId> | <ParameterRef>)*;

//Implementations of different types of attributes appearing in XSD
<ParameterName; ArrayId = "Name"; ValueRef = "Name"; Type = WString> :=
```

```
"name" ~"=" <$Name>;
<ParameterId; ArrayId = "ID"; ValueRef = "Name"; Type = WString> :=
"id" ~"=" <$Name>;
<ParameterRef; ArrayId = "Reference"; ValueRef = "URL"; Type = WString> :=
"ref" ~"=" <$URL>;
<BaseParameter; ArrayId = "Base"; ValueRef = "URL"; Type = WString> :=
"base" ~"=" <$URL>;
<ParameterType; ArrayId = "Type"; ValueRef = "URL" ; Type = WString> :=
"type" ~"=" <$URL>;
<ParameterDefault; ArrayId = "Default"; ValueRef = "DefaultFixedString" ; Type = WString> :=
"default" ~"=" <$DefaultFixedString>;
<ParameterFixed; ArrayId = "Fixed"; ValueRef = "DefaultFixedString"; Type = WString> :=
"fixed" ~"=" <$DefaultFixedString>;
<ParameterMaxOccurs; ArrayId = "MaxOccurs"; ValueRef = "OccursValue"> :=
"maxOccurs" ~"=" <OccursValue>;
<ParameterMinOccurs; ArrayId = "MinOccurs"; ValueRef = "OccursValue"> :=
"minOccurs" ~"=" <OccursValue>;
<OccursValue; Type = WString> := <$Integer> | "\"unbounded\"";
<ParameterUse; ArrayId = "Use"; ValueRef = "UseValues"> :=
"use" ~"=" <UseValues>;
<UseValues; Type = WString> := "\"required\"" | "\"optional\"";
<ParameterNamespace; ArrayId = "Namespace"; ValueRef = "Namespace"; Type = WString> :=
"namespace" ~"=" <$Namespace>;
<ParameterSchemaLocation; ArrayId = "Location"; ValueRef = "Namespace"; Type = WString> :=
"schemaLocation" ~"=" <$Namespace>;
<ParameterXPath; ArrayId = "XPath"; ValueRef = "URL"; Type = WString> :=
"xpath" ~"=" <$URL>;
<ParameterProcessContents; ArrayId = "ProcessContents"; ValueRef = "Name"; Type = WString> :=
"processContents" ~"=" <$Name>;
<ParameterMixed; ArrayId = "Mixed"; Type = WString> :=
~"mixed" ~"=" ("\"true\"" | "\"false\"");
<ParameterAbstract; ArrayId = "Abstract"; Type = WString> :=
~"abstract" ~"=" ("\"true\"" | "\"false\"");
<ParameterItemType; ArrayId = "ItemType"; ValueRef = "URL"; Type = WString> :=
~"itemType" ~"=" <$URL>;
<ParameterMemberType; ArrayId = "MemberTypes"; ValueRef = "URL"; Type = WString> :=
~"memberTypes" ~"=" <$URL>;
<ParameterSubstitutionGroup; ArrayId = "SubstitutionGroup"; ValueRef = "URL"; Type = WString> :=
~"substitutionGroup" ~"=" <$URL>;
<ParameterBlock; ArrayId = "Block"; ValueRef = "URL"; Type = WString> :=
~"block" ~"=" <$URL>;
<ParameterNillable; ArrayId = "Nillable"; Type = WString> :=
~"nillable" ~"=" ("\"true\"" | "\"false\"");
<ParameterForm; ArrayId = "Form"; Type = WString> :=
~"form" ~"=" ("\"qualified\"" | "\"unqualified\"");
<ParameterFinal; ArrayId = "Form"; ValueRef = "URL"; Type = WString> :=
~"final" ~"=" <$url>;
//End attribute logic

//End element logic
//End non-terminals
```

B Example XSD

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:appinfo>sgm.xsd v0.1 2003-01</xsd:appinfo>
    <xsd:documentation>Copyright (c) 2003-2014 Schut Geometrical Metrology.</xsd:documentation>
  </xsd:annotation>

  <xsd:element name="data" type="dataType"/>

  <xsd:complexType name="dataType">
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="arr" type="arrayType"/>
        <xsd:element name="sto" type="storableType"/>
        <xsd:element name="cus" type="customizableType"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attributeGroup ref="dataTypeAttributes"/>
  </xsd:complexType>

  <xsd:complexType name="arrayBaseType">
    <xsd:sequence>
      <xsd:element name="elem" type="arrayElementType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="addr" type="addressType" use="required"/>
    <xsd:attribute name="size" type="xsd:unsignedInt" use="required"/>
    <xsd:attribute name="duplicateIDs" type="xsd:boolean" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="arrayType">
    <xsd:complexContent>
      <xsd:extension base="arrayBaseType">
        <xsd:attribute name="uid" type="uidType" use="optional"/>
        <xsd:attribute name="bwp" type="addressType" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="storableType">
    <xsd:complexContent>
      <xsd:extension base="arrayBaseType">
        <xsd:attribute name="uid" type="uidType" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="customizableType">
    <xsd:complexContent>
      <xsd:extension base="arrayBaseType">
```

```
      <xsd:attribute name="uid" type="uidType" use="required"/>
      <xsd:attribute name="bwp" type="addressType" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="arrayElementType">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:string" minOccurs="0"/>
    <xsd:choice>
      <xsd:element name="nil" type="nilValueType"/>
      <xsd:element name="bool" type="boolValueType"/>
      <xsd:element name="char" type="charValueType"/>
      <xsd:element name="uchar" type="ucharValueType"/>
      <xsd:element name="wchar" type="wcharValueType"/>
      <xsd:element name="short" type="shortValueType"/>
      <xsd:element name="ushort" type="ushortValueType"/>
      <xsd:element name="long" type="longValueType"/>
      <xsd:element name="ulong" type="ulongValueType"/>
      <xsd:element name="int" type="intValueType"/>
      <xsd:element name="uint" type="uintValueType"/>
      <xsd:element name="float" type="floatValueType"/>
      <xsd:element name="double" type="doubleValueType"/>
      <xsd:element name="string" type="stringValueType"/>
      <xsd:element name="wstring" type="wstringValueType"/>
      <xsd:element name="array" type="arrayValueType"/>
      <xsd:element name="storable" type="storableValueType"/>
      <xsd:element name="cust" type="customizableValueType"/>
      <xsd:element name="reference" type="referenceValueType"/>
      <xsd:element name="multidouble" type="multiDoubleValueType"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="idx" type="xsd:unsignedInt" use="required"/>
</xsd:complexType>

<xsd:attributeGroup name="valueTypeAttributes">
  <xsd:attribute name="flags" type="xsd:short" use="optional"/>
</xsd:attributeGroup>

<xsd:simpleType name="vrmType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="([1-9][0-9]*|0)\.[0-9][1-9]?\.[0-9]{1,4}(-([1-9][0-9]*))?"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:attributeGroup name="dataTypeAttributes">
  <xsd:attribute name="ver" type="vrmType" use="required"/>
  <xsd:attribute name="scd" type="vrmType" use="required"/>
  <xsd:attribute name="root" type="addressType" use="required"/>
</xsd:attributeGroup>

<xsd:simpleType name="doubleType">
  <xsd:restriction base="xsd:double">
    <xsd:minInclusive value="-1.7976931348623158e+308"/>
```

```
      <xsd:maxInclusive value="1.7976931348623158e+308"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="floatType">
    <xsd:restriction base="xsd:float">
      <xsd:minInclusive value="-3.402823466e+38"/>
      <xsd:maxInclusive value="3.402823466e+38"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="addressType">
    <xsd:restriction base="xsd:unsignedInt"/>
  </xsd:simpleType>

  <xsd:simpleType name="uidType">
    <xsd:restriction base="xsd:unsignedInt"/>
  </xsd:simpleType>

  <xsd:complexType name="pathType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="sep" type="xsd:string" use="optional"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:attributeGroup name="quantifiedValueTypeAttributes">
    <xsd:attributeGroup ref="valueTypeAttributes"/>
    <xsd:attribute name="qty" type="xsd:unsignedInt" use="optional"/>
  </xsd:attributeGroup>

  <xsd:complexType name="typeInfoType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:unsignedInt">
        <xsd:attribute name="qty" type="xsd:unsignedInt" use="optional"/>
        <xsd:attribute name="uid" type="uidType" use="optional"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="boolValueType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:boolean">
        <xsd:attributeGroup ref="valueTypeAttributes"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="charValueType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:byte">
        <xsd:attributeGroup ref="valueTypeAttributes"/>
      </xsd:extension>
```

```
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="ucharValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedByte">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="wcharValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedShort">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="shortValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:short">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="ushortValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedShort">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="longValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:int">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="ulongValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedInt">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="nilValueType">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
```

```
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="intValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:int">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="uintValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedInt">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="floatValueType">
  <xsd:simpleContent>
    <xsd:extension base="floatType">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="doubleValueType">
  <xsd:simpleContent>
    <xsd:extension base="doubleType">
      <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="stringValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="wstringValueType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="arrayValueType">
```

```
  <xsd:simpleContent>
    <xsd:extension base="addressType">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="storableValueType">
  <xsd:simpleContent>
    <xsd:extension base="addressType">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="customizableValueType">
  <xsd:simpleContent>
    <xsd:extension base="addressType">
      <xsd:attributeGroup ref="valueTypeAttributes"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="referenceValueType">
  <xsd:sequence>
    <xsd:element name="base" type="pathType"/>
    <xsd:element name="item" type="pathType"/>
    <xsd:element name="type" type="typeInfoType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="lnk" type="xsd:boolean" use="required"/>
  <xsd:attribute name="usg" type="xsd:unsignedByte" use="required"/>
</xsd:complexType>

<xsd:simpleType name="dimensionType">
  <xsd:restriction base="xsd:unsignedInt"/>
</xsd:simpleType>

<xsd:complexType name="boundsType">
  <xsd:sequence>
    <xsd:element name="dim" type="dimensionType" minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="size" type="xsd:unsignedInt" use="required"/>
</xsd:complexType>

<xsd:simpleType name="doubleVectorValueType">
  <xsd:restriction base="doubleType"/>
</xsd:simpleType>

<xsd:complexType name="doubleVectorType">
  <xsd:sequence>
    <xsd:element name="dval" type="doubleVectorValueType" minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="size" type="xsd:unsignedInt" use="required"/>
</xsd:complexType>
```

```
  <xsd:complexType name="multiDoubleValueType">
    <xsd:sequence>
      <xsd:element name="bnds" type="boundsType"/>
      <xsd:element name="dvec" type="doubleVectorType"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="quantifiedValueTypeAttributes"/>
  </xsd:complexType>

</xsd:schema>
```

C UBNF of example XSD

```
//Metadata
( Start Rule ) := <Root>;
( Single Line Comment ) := "//";
( Multi Line Comment ) := "<!--" "-->";
( Keep Comments ) := false;
//End metadata

//Terminals
{StringCh} := {%AlphaNumeric} + [*+%_-?!@#$=\\.,:;^$&|\[\]{}()];
{Base64Char} := {%AlphaNumeric} + [+/=];

<$word> := {StringCh}+;
<$base64word> := {Base64Char}+;
<$string> := <$word> (" "+ <$word>)*;
<$positiveInteger> := {%Digit}+ ("e" ("+" | "-") {%Digit}+)?;
<$negativeInteger> := "-" {%Digit}+ ("e" ("+" | "-") {%Digit}+)?;
<$integer> := "-"? {%Digit}+ ("e" ("+" | "-") {%Digit}+)?;
<$decimal> := "-"? {%Digit}+ ("." {%Digit}+)? ("e" ("+" | "-") {%Digit}+)?;
<$positiveDecimal> := {%Digit}+ ("." {%Digit}+)? ("e" ("+" | "-") {%Digit}+)?;
<$negativeDecimal> := "-" {%Digit}+ ("." {%Digit}+)? ("e" ("+" | "-") {%Digit}+)?;
<boolean; Alias=true> := "true" | "false";
<$VersionString> := {%Number}+ "." {%Number}+;
<$base64> := <$base64word> (" "+ <$base64word>)*;
//End terminals

//Start non-terminals
<Root> := <XML_Version>? <Schema>;

<XML_Version; ArrayId = "Metadata"> := ~"<?xml" <Version_String> <Encoding_String>? ~"?>";
<Version_String; ValueRef = "VersionString"; ArrayId = "Version"> := "version=" "\"" <$VersionString> "\"";
<Encoding_String; ValueRef = "string"; ArrayId = "Encoding"> := "encoding=" "\"" <$string> "\"";

<Any_Attribute; ArrayIdRef="word"; ValueRef="string"> := <$word> ~"=" ~"\"" <$string> ~"\"";
<Any_Core; ArrayIdRef="word"> :=
~"<" <$word> <Any_Attribute>* ~">"
(<$string> | <Any_Core>*)
~"</" <$word> ~">";

<Schema> := <data>;

<id_Content; ArrayId="Content"; Type=WString> := <$string>;
<id; Alias=true> := <id_Core>?;
<id_Core; ArrayId="id"> := <Simple_id> | <Complex_id>;
<Simple_id; Alias=true> := ~"<id/>";
<Complex_id; Alias=true> :=
~"<id>"
<id_Content>?
~"</id>";

<bool_Attribute_flags; Type=Short; ArrayId="flags"> := ~"flags=" ~"\"" <$integer> ~"\"";
<nil_Content; ArrayId="Content"; Type=WString> := <$string>;
```

```
<nil; ArrayId="nil"> := <Simple_nil> | <Complex_nil>;
<Simple_nil; Alias=true> := ~"<nil" (<bool_Attribute_flags>)* ~"/>";
<Complex_nil; Alias=true> :=
~"<nil" (<bool_Attribute_flags>)* ~">"
<nil_Content>?
~"</nil>";

<bool_Content; ArrayId="Content"; Type=Bool> := <boolean>;
<bool; ArrayId="bool"> := <Simple_bool> | <Complex_bool>;
<Simple_bool; Alias=true> := ~"<bool" (<bool_Attribute_flags>)* ~"/>";
<Complex_bool; Alias=true> :=
~"<bool" (<bool_Attribute_flags>)* ~">"
<bool_Content>?
~"</bool>";

<char_Content; ArrayId="Content"; Type=UChar> := <$integer>;
<char; ArrayId="char"> := <Simple_char> | <Complex_char>;
<Simple_char; Alias=true> := ~"<char" (<bool_Attribute_flags>)* ~"/>";
<Complex_char; Alias=true> :=
~"<char" (<bool_Attribute_flags>)* ~">"
<char_Content>?
~"</char>";

<uchar_Content; ArrayId="Content"; Type=UChar> := <$positiveInteger>;
<uchar; ArrayId="uchar"> := <Simple_uchar> | <Complex_uchar>;
<Simple_uchar; Alias=true> := ~"<uchar" (<bool_Attribute_flags>)* ~"/>";
<Complex_uchar; Alias=true> :=
~"<uchar" (<bool_Attribute_flags>)* ~">"
<uchar_Content>?
~"</uchar>";

<wchar_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<wchar; ArrayId="wchar"> := <Simple_wchar> | <Complex_wchar>;
<Simple_wchar; Alias=true> := ~"<wchar" (<bool_Attribute_flags>)* ~"/>";
<Complex_wchar; Alias=true> :=
~"<wchar" (<bool_Attribute_flags>)* ~">"
<wchar_Content>?
~"</wchar>";

<short_Attribute_qty; Type=UInt; ArrayId="qty"> := ~"qty=" ~"\"" <$positiveInteger> ~"\"";
<short_Content; ArrayId="Content"; Type=Short> := <$integer>;
<short; ArrayId="short"> := <Simple_short> | <Complex_short>;
<Simple_short; Alias=true> := ~"<short" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_short; Alias=true> :=
~"<short" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<short_Content>?
~"</short>";

<ushort_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<ushort; ArrayId="ushort"> := <Simple_ushort> | <Complex_ushort>;
<Simple_ushort; Alias=true> := ~"<ushort" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_ushort; Alias=true> :=
~"<ushort" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<ushort_Content>?
```

~"</ushort>";

<long_Content; ArrayId="Content"; Type=Int> := <$integer>;
<long; ArrayId="long"> := <Simple_long> | <Complex_long>;
<Simple_long; Alias=true> := ~"<long" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_long; Alias=true> :=
~"<long" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<long_Content>?
~"</long>";

<ulong_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<ulong; ArrayId="ulong"> := <Simple_ulong> | <Complex_ulong>;
<Simple_ulong; Alias=true> := ~"<ulong" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_ulong; Alias=true> :=
~"<ulong" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<ulong_Content>?
~"</ulong>";

<int_Content; ArrayId="Content"; Type=Int> := <$integer>;
<int; ArrayId="int"> := <Simple_int> | <Complex_int>;
<Simple_int; Alias=true> := ~"<int" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_int; Alias=true> :=
~"<int" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<int_Content>?
~"</int>";

<uint_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<uint; ArrayId="uint"> := <Simple_uint> | <Complex_uint>;
<Simple_uint; Alias=true> := ~"<uint" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_uint; Alias=true> :=
~"<uint" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<uint_Content>?
~"</uint>";

<float_Content; ArrayId="Content"; Type=Float;
MaxInclusive=3.402823466e+38; MinInclusive=−3.402823466e+38> := <$decimal>;
<float; ArrayId="float"> := <Simple_float> | <Complex_float>;
<Simple_float; Alias=true> := ~"<float/>";
<Complex_float; Alias=true> :=
~"<float>"
<float_Content>?
~"</float>";

<double_Content; ArrayId="Content"; Type=Float;
MaxInclusive=1.7976931348623158e+308; MinInclusive=−1.7976931348623158e+308> := <$decimal>;
<double; ArrayId="double"> := <Simple_double> | <Complex_double>;
<Simple_double; Alias=true> := ~"<double/>";
<Complex_double; Alias=true> :=
~"<double>"
<double_Content>?
~"</double>";

<string_Content; ArrayId="Content"; Type=WString> := <$string>;
<string; ArrayId="string"> := <Simple_string> | <Complex_string>;

```
<Simple_string; Alias=true> := ~"<string" (<bool_Attribute_flags>)* ~"/>";
<Complex_string; Alias=true> :=
~"<string" (<bool_Attribute_flags>)* ~">"
<string_Content>?
~"</string>";

<wstring_Content; ArrayId="Content"; Type=WString> := <$string>;
<wstring; ArrayId="wstring"> := <Simple_wstring> | <Complex_wstring>;
<Simple_wstring; Alias=true> := ~"<wstring" (<bool_Attribute_flags>)* ~"/>";
<Complex_wstring; Alias=true> :=
~"<wstring" (<bool_Attribute_flags>)* ~">"
<wstring_Content>?
~"</wstring>";

<array_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<array; ArrayId="array"> := <Simple_array> | <Complex_array>;
<Simple_array; Alias=true> := ~"<array/>";
<Complex_array; Alias=true> :=
~"<array>"
<array_Content>?
~"</array>";

<storable_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<storable; ArrayId="storable"> := <Simple_storable> | <Complex_storable>;
<Simple_storable; Alias=true> := ~"<storable/>";
<Complex_storable; Alias=true> :=
~"<storable>"
<storable_Content>?
~"</storable>";

<cust_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<cust; ArrayId="cust"> := <Simple_cust> | <Complex_cust>;
<Simple_cust; Alias=true> := ~"<cust/>";
<Complex_cust; Alias=true> :=
~"<cust>"
<cust_Content>?
~"</cust>";

<pathType_Attribute_sep; Type=WString; ArrayId="sep"> := ~"sep=" ~"\"" <$string> ~"\"";
<base_Content; ArrayId="Content"; Type=WString> := <$string>;
<base; ArrayId="base"> := <Simple_base> | <Complex_base>;
<Simple_base; Alias=true> := ~"<base" (<pathType_Attribute_sep>)* ~"/>";
<Complex_base; Alias=true> :=
~"<base" (<pathType_Attribute_sep>)* ~">"
<base_Content>?
~"</base>";

<item_Content; ArrayId="Content"; Type=WString> := <$string>;
<item; ArrayId="item"> := <Simple_item> | <Complex_item>;
<Simple_item; Alias=true> := ~"<item" (<pathType_Attribute_sep>)* ~"/>";
<Complex_item; Alias=true> :=
~"<item" (<pathType_Attribute_sep>)* ~">"
<item_Content>?
~"</item>";
```

<typeInfoType_Attribute_qty; Type=UInt; ArrayId="qty"> := ~"qty=" ~"\"" <$positiveInteger> ~"\"";
<typeInfoType_Attribute_uid; Type=UInt; ArrayId="uid"> := ~"uid=" ~"\"" <$positiveInteger> ~"\"";
<type_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<type; Alias=true> := <type_Core>?;
<type_Core; ArrayId="type"> := <Simple_type> | <Complex_type>;
<Simple_type; Alias=true> := ~"<type" (<typeInfoType_Attribute_qty> | <typeInfoType_Attribute_uid>)* ~"/>";
<Complex_type; Alias=true> :=
~"<type" (<typeInfoType_Attribute_qty> | <typeInfoType_Attribute_uid>)* ~">"
<type_Content>?
~"</type>";

<Sequence6; Alias=true> := <base> <item> <type>?;

<referenceValueType_Attribute_lnk; Type=Bool; ArrayId="lnk"> := ~"lnk=" ~"\"" <boolean> ~"\"";
<referenceValueType_Attribute_usg; Type=UChar; ArrayId="usg"> := ~"usg=" ~"\"" <$positiveInteger> ~"\"";
<reference; ArrayId="reference"> := <Simple_reference> | <Complex_reference>;
<Simple_reference; Alias=true> := ~"<reference"
(<referenceValueType_Attribute_lnk> | <referenceValueType_Attribute_usg>)* ~"/>";
<Complex_reference; Alias=true> :=
~"<reference" (<referenceValueType_Attribute_lnk> | <referenceValueType_Attribute_usg>)* ~">"
<Sequence6>
~"</reference>";

<dim_Content; ArrayId="Content"; Type=UInt> := <$positiveInteger>;
<dim; Alias=true> := <dim_Core> <dim_Core>*;
<dim_Core; ArrayId="dim"> := <Simple_dim> | <Complex_dim>;
<Simple_dim; Alias=true> := ~"<dim/>";
<Complex_dim; Alias=true> :=
~"<dim>"
<dim_Content>?
~"</dim>";

<Sequence7; Alias=true> := <dim>;

<boundsType_Attribute_size; Type=UInt; ArrayId="size"> := ~"size=" ~"\"" <$positiveInteger> ~"\"";
<bnds; ArrayId="bnds"> := <Simple_bnds> | <Complex_bnds>;
<Simple_bnds; Alias=true> := ~"<bnds" (<boundsType_Attribute_size>)* ~"/>";
<Complex_bnds; Alias=true> :=
~"<bnds" (<boundsType_Attribute_size>)* ~">"
<Sequence7>
~"</bnds>";

<dval_Content; ArrayId="Content"; Type=Float;
MaxInclusive=1.7976931348623158e+308; MinInclusive=−1.7976931348623158e+308> := <$decimal>;
<dval; Alias=true> := <dval_Core> <dval_Core>*;
<dval_Core; ArrayId="dval"> := <Simple_dval> | <Complex_dval>;
<Simple_dval; Alias=true> := ~"<dval/>";
<Complex_dval; Alias=true> :=
~"<dval>"
<dval_Content>?
~"</dval>";

<Sequence8; Alias=true> := <dval>;

<doubleVectorType_Attribute_size; Type=UInt; ArrayId="size"> := ~"size=" ~"\"" <$positiveInteger> ~"\"";
<dvec; ArrayId="dvec"> := <Simple_dvec> | <Complex_dvec>;
<Simple_dvec; Alias=true> := ~"<dvec" (<doubleVectorType_Attribute_size>)* ~"/>";
<Complex_dvec; Alias=true> :=
~"<dvec" (<doubleVectorType_Attribute_size>)* ~">"
<Sequence8>
~"</dvec>";

<Sequence9; Alias=true> := <bnds> <dvec>;

<multidouble; ArrayId="multidouble"> := <Simple_multidouble> | <Complex_multidouble>;
<Simple_multidouble; Alias=true> := ~"<multidouble" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~"/>";
<Complex_multidouble; Alias=true> :=
~"<multidouble" (<short_Attribute_qty> | <bool_Attribute_flags>)* ~">"
<Sequence9>
~"</multidouble>";

<Choice4; Alias=true> := <nil> | <bool> | <char> | <uchar> | <wchar> | <short> | <ushort> |
<long> | <ulong> | <int> | <uint> | <float> | <double> | <string> | <wstring> | <array> |
<storable> | <cust> | <reference> | <multidouble>;

<Sequence5; Alias=true> := <id>? <Choice4>;

<arrayElementType_Attribute_idx; Type=UInt; ArrayId="idx"> := ~"idx=" ~"\"" <$positiveInteger> ~"\"";
<elem; Alias=true> := <elem_Core>*;
<elem_Core; ArrayId="elem"> := <Simple_elem> | <Complex_elem>;
<Simple_elem; Alias=true> := ~"<elem" (<arrayElementType_Attribute_idx>)* ~"/>";
<Complex_elem; Alias=true> :=
~"<elem" (<arrayElementType_Attribute_idx>)* ~">"
<Sequence5>
~"</elem>";

<Sequence3; Alias=true> := <elem>?;

<arrayType_Attribute_uid; Type=UInt; ArrayId="uid"> := ~"uid=" ~"\"" <$positiveInteger> ~"\"";
<arrayType_Attribute_bwp; Type=UInt; ArrayId="bwp"> := ~"bwp=" ~"\"" <$positiveInteger> ~"\"";
<arrayBaseType_Attribute_addr; Type=UInt; ArrayId="addr"> := ~"addr=" ~"\"" <$positiveInteger> ~"\"";
<arrayBaseType_Attribute_size; Type=UInt; ArrayId="size"> := ~"size=" ~"\"" <$positiveInteger> ~"\"";
<arrayBaseType_Attribute_duplicateIDs; Type=Bool; ArrayId="duplicateIDs"> := ~"duplicateIDs=" ~"\"" <boolean> ~"\"";
<arr; ArrayId="arr"> := <Simple_arr> | <Complex_arr>;
<Simple_arr; Alias=true> := ~"<arr" (<arrayType_Attribute_uid> | <arrayType_Attribute_bwp> |
<arrayBaseType_Attribute_addr> | <arrayBaseType_Attribute_size> |
<arrayBaseType_Attribute_duplicateIDs>)* ~"/>";
<Complex_arr; Alias=true> :=
~"<arr" (<arrayType_Attribute_uid> | <arrayType_Attribute_bwp> | <arrayBaseType_Attribute_addr> |
<arrayBaseType_Attribute_size> | <arrayBaseType_Attribute_duplicateIDs>)* ~">"
<Sequence3>
~"</arr>";

<storableType_Attribute_uid; Type=UInt; ArrayId="uid"> := ~"uid=" ~"\"" <$positiveInteger> ~"\"";
<sto; ArrayId="sto"> := <Simple_sto> | <Complex_sto>;
<Simple_sto; Alias=true> := ~"<sto" (<storableType_Attribute_uid> | <arrayBaseType_Attribute_addr> |
<arrayBaseType_Attribute_size> | <arrayBaseType_Attribute_duplicateIDs>)* ~"/>";

```
<Complex_sto; Alias=true> :=
~"<sto" (<storableType_Attribute_uid> | <arrayBaseType_Attribute_addr> | <arrayBaseType_Attribute_size> |
<arrayBaseType_Attribute_duplicateIDs>)* ~">"
<Sequence3>
~"</sto>";

<customizableType_Attribute_uid; Type=UInt; ArrayId="uid"> := ~"uid=" ~"\"" <$positiveInteger> ~"\"";
<customizableType_Attribute_bwp; Type=UInt; ArrayId="bwp"> := ~"bwp=" ~"\"" <$positiveInteger> ~"\"";
<cus; ArrayId="cus"> := <Simple_cus> | <Complex_cus>;
<Simple_cus; Alias=true> := ~"<cus" (<customizableType_Attribute_uid> | <customizableType_Attribute_bwp> |
<arrayBaseType_Attribute_addr> | <arrayBaseType_Attribute_size> | <arrayBaseType_Attribute_duplicateIDs>)* ~"/>";
<Complex_cus; Alias=true> :=
~"<cus" (<customizableType_Attribute_uid> | <customizableType_Attribute_bwp> | <arrayBaseType_Attribute_addr>
| <arrayBaseType_Attribute_size> | <arrayBaseType_Attribute_duplicateIDs>)* ~">"
<Sequence3>
~"</cus>";

<Choice1; Alias=true> := <Choice1_Core>*;
<Choice1_Core; Alias=true> := <arr> | <sto> | <cus>;

<Sequence2; Alias=true> := <Choice1>?;

<data_Attribute_ver; Type=WString; ArrayId="ver"> := ~"ver=" ~"\"" <$string> ~"\"";
<data_Attribute_scd; Type=WString; ArrayId="scd"> := ~"scd=" ~"\"" <$string> ~"\"";
<data_Attribute_root; Type=UInt; ArrayId="root"> := ~"root=" ~"\"" <$positiveInteger> ~"\"";
<data; ArrayId="data"> := <Simple_data> | <Complex_data>;
<Simple_data; Alias=true> := ~"<data" (<data_Attribute_ver> | <data_Attribute_scd> | <data_Attribute_root>)* ~"/>";
<Complex_data; Alias=true> :=
~"<data" (<data_Attribute_ver> | <data_Attribute_scd> | <data_Attribute_root>)* ~">"
<Sequence2>
~"</data>";

//End non-terminals
```

D Individual results of performance testing

Section 8.3.2 presents results as an average of 5 test runs. This section notates every one of the 5 test runs for each project to ensure full clarity of the results.

Note that type requests and available types, both shown in the performance analysis overview, will always remain the same in each run, as such, they are not displayed within the appendix.

Purchase Order run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	7.3 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1559 ms	1044.6 ms		2611 ms

Figure 102: First run on the Purchase Order XSD project

Purchase Order run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	8.8 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1547 ms	1019.4 ms		2575.3 ms

Figure 103: Second run on the Purchase Order XSD project

Purchase Order run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	8.6 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1565.4 ms	1035.4 ms		2609.5 ms

Figure 104: Third run on the Purchase Order XSD project

Purchase Order run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	9.7 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1544.9 ms	1057.4 ms		2612.1 ms

Figure 105: Fourth run on the Purchase Order XSD project

Purchase Order run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	8.9 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1610.8 ms	1022.8 ms		2642.6 ms

Figure 106: Fifth run on the Purchase Order XSD project

IBM Substitution run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	7.7 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1183.3 ms	735.1 ms		1926.2 ms

Figure 107: First run on the IBM Substitution XSD project

IBM Substitution run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	8.7 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1197 ms	716.3 ms		1922.1 ms

Figure 108: Second run on the IBM Substitution XSD project

IBM Substitution run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	10.3 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1150.3 ms	747.7 ms		1908.4 ms

Figure 109: Third run on the IBM Substitution XSD project

IBM Substitution run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	10.4 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1158.7 ms	719.7 ms		1888.9 ms

Figure 110: Fourth run on the IBM Substitution XSD project

IBM Substitution run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	10.4 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1169.7 ms	796.8 ms		1977 ms

Figure 111: Fifth run on the IBM Substitution XSD project

IBM Testsuite run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	12 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1503.8 ms	952.5 ms		2468.4 ms

Figure 112: First run on the IBM Testsuite XSD project

IBM Testsuite run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	12.4 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1493.4 ms	946.7 ms		2452.6 ms

Figure 113: Second run on the IBM Testsuite XSD project

IBM Testsuite run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	11.2 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1537.2 ms	933.4 ms		2481.9 ms

Figure 114: Third run on the IBM Testsuite XSD project

IBM Testsuite run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	12.4 ms	0.2 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1508.7 ms	982.5 ms		2503.8 ms

Figure 115: Fourth run on the IBM Testsuite XSD project

IBM Testsuite run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	9.4 ms	0.1 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
1645 ms	1084.4 ms		2738.9 ms

Figure 116: Fifth run on the IBM Testsuite XSD project

SGM run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	29.4 ms	0.4 ms	0.1 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2525.8 ms	2930.1 ms		5485.8 ms

Figure 117: First run on the SGM XSD project

SGM run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	30.6 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2529.8 ms	2957.1 ms		5517.9 ms

Figure 118: Second run on the SGM XSD project

SGM run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	37.2 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2593.1 ms	2955.4 ms		5586.1 ms

Figure 119: Third run on the SGM XSD project

SGM run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	37.3 ms	0.5 ms	0.1 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2527.1 ms	2952.8 ms		5517.8 ms

Figure 120: Fourth run on the SGM XSD project

SGM run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	30.4 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2518.9 ms	2933.2 ms		5482.9 ms

Figure 121: Fifth run on the SGM XSD project

Thales Web Service run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	48 ms	0.5 ms	0.1 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
25761.1 ms	9567.1 ms		35376.8 ms

Figure 122: First run on the Thales Web Service XSD project

Thales Web Service run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	43.6 ms	0.5 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
27863.5 ms	9642.7 ms		37550.3 ms

Figure 123: Second run on the Thales Web Service XSD project

Thales Web Service run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	54.3 ms	0.5 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
25728.5 ms	9511.8 ms		35295.1 ms

Figure 124: Third run on the Thales Web Service XSD project

Thales Web Service run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	53.6 ms	0.5 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
26273.1 ms	9572.9 ms		35900.1 ms

Figure 125: Fourth run on the Thales Web Service XSD project

Thales Web Service run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	44.9 ms	0.5 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
26058.8 ms	9836.5 ms		35940.7 ms

Figure 126: Fifth run on the Thales Web Service XSD project

Thales Activation run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	35.2 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2310 ms	1769.4 ms		4115 ms

Figure 127: First run on the Thales Activation XSD project

Thales Activation run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	42.9 ms	0.3 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2459.5 ms	1880.1 ms		4382.8 ms

Figure 128: Second run on the Thales Activation XSD project

Thales Activation run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	36.2 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2397.3 ms	1816.4 ms		4250.3 ms

Figure 129: Third run on the Thales Activation XSD project

Thales Activation run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	43.6 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2395.8 ms	1851.8 ms		4291.6 ms

Figure 130: Fourth run on the Thales Activation XSD project

Thales Activation run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	40.6 ms	0.4 ms	0 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2480.6 ms	1864.5 ms		4386.1 ms

Figure 131: Fifth run on the Thales Activation XSD project

SEPA run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
0 ms	101 ms	1.6 ms	0.4 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
4810.5 ms	5619.3 ms		10532.8 ms

Figure 132: First run on the SEPA XSD project

SEPA run #2			
Pre-processor time: 0 ms	Parser time: 103.3 ms	Converter time: 1.5 ms	Post-processor time: 0.4 ms
Terminal state builder time: 5142.8 ms	Grammar state builder time: 5664.9 ms	Total execution time: 10912.9 ms	

Figure 133: Second run on the SEPA XSD project

SEPA run #3			
Pre-processor time: 0 ms	Parser time: 103.4 ms	Converter time: 1.6 ms	Post-processor time: 0.4 ms
Terminal state builder time: 4914.6 ms	Grammar state builder time: 5558 ms	Total execution time: 10578 ms	

Figure 134: Third run on the SEPA XSD project

SEPA run #4			
Pre-processor time: 0 ms	Parser time: 104.8 ms	Converter time: 1.5 ms	Post-processor time: 0.4 ms
Terminal state builder time: 4950 ms	Grammar state builder time: 5637.8 ms	Total execution time: 10694.5 ms	

Figure 135: Fourth run on the SEPA XSD project

SEPA run #5			
Pre-processor time: 0 ms	Parser time: 102.8 ms	Converter time: 1.7 ms	Post-processor time: 0.4 ms
Terminal state builder time: 4842.5 ms	Grammar state builder time: 5569.8 ms	Total execution time: 10517.2 ms	

Figure 136: Fifth run on the SEPA XSD project

UBL run #1			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1690 ms	17.1 ms	16.3 ms	21.2 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2821485.4 ms	1324451.9 ms		4147681.9 ms

Figure 137: First run on the UBL XSD project

UBL run #2			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1988.1 ms	17.7 ms	20.5 ms	15 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2822111.3 ms	1324512.1 ms		4148664.7 ms

Figure 138: Second run on the UBL XSD project

UBL run #3			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1657.1 ms	16.3 ms	19.4 ms	14.7 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2821998.4 ms	1323512.3 ms		4147218.2 ms

Figure 139: Third run on the UBL XSD project

UBL run #4			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1677.5 ms	16.9 ms	18.7 ms	14.8 ms
Terminal state builder time:	Grammar state builder time:		Total execution time:
2822418 ms	1322891.2 ms		4147037.1 ms

Figure 140: Fourth run on the UBL XSD project

UBL run #5			
Pre-processor time:	Parser time:	Converter time:	Post-processor time:
1745.5 ms	24.1 ms	19.3 ms	14.7 ms
Terminal state builder time:	Grammar state builder time:	Total execution time:	
2825312.9 ms	1324412.1 ms	4151528.6 ms	

Figure 141: Fifth run on the UBL XSD project