Creating a readable language for checking XML

Daniel Forsman
Creating a readable language for checking XML

Daniel Forsman
atp10dfo@student.lu.se

June 15, 2015

Master’s thesis work carried out at Advenica.

Supervisors: Niklas Fors, niklas.fors@cs.lth.se
Alexander Nilsson, alexander.nilsson@advenica.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se
Abstract

Today sharing data is done everywhere. Doctors might want to share patient journal information. Patient journals may contain sensitive information that doctors do not want to share. The journals need to be checked before they are shared. In this thesis, data and journals are coded in XML and checking journals and data is the same as validating XML.

Validating XML documents is usually done by following rules from a validator. A validator processes XML documents and checks that the XML documents follow the validation rules. The issue with most validators today is that they cannot compare arbitrary elements in the XML document with each other and there are no mathematical operations to supply these comparisons.

Sometimes there is a need to verify the validation rules. This might be done by someone who has little programming skills. The validator has to be readable so that this someone can verify that the validator matches the requirements.

This thesis attempts to solve the issue with existing solutions by creating a readable language for validating XML documents. The solution is done in three steps: investigating similar solutions, implementing a validator, and testing the readability of the validator with a usability test.

Keywords: XML, XPath, validation language, readability, JastAdd
Acknowledgements

I wish to thank Alexander Nilsson and Niklas Fors for giving me valuable advice and guidance during this thesis. A big thank you to Advenica for providing me with a computer and breakfast. Victor Johnsson and Anders Tilly I thank for their opposition of my thesis.
# Contents

## 1 Introduction

1.1 Motivation ........................................ 9  
  1.1.1 Contribution .................................. 10  
1.2 Methodology ......................................... 11  
  1.2.1 Useful resources ................................ 11  
  1.2.2 Creating a validation language ................. 11  
  1.2.3 Testing readability ............................... 11

## 2 Architecture

2.1 Validator definition ................................ 13  
2.2 Validator construction ............................... 13  
2.3 Test case ............................................. 14  
  2.3.1 Computing the distance between coordinates .... 15

## 3 Background

3.1 Compiler ............................................. 17  
  3.1.1 JastAdd ........................................... 17  
  3.1.2 Scanner ............................................ 18  
  3.1.3 Parser .............................................. 18  
  3.1.4 Formal grammar ................................... 19  
3.2 Data structure ....................................... 20  
  3.2.1 XML ................................................ 20  
  3.2.2 JSON .............................................. 21  
  3.2.3 Example ........................................... 21  
3.3 Choosing data format ............................... 22  
3.4 XML validation ....................................... 22  
  3.4.1 Example ........................................... 23  
3.5 XPath ................................................ 23  
3.6 XSLT .................................................. 24  
3.7 Parsing XML .......................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.1 DOM</td>
<td>25</td>
</tr>
<tr>
<td>3.7.2 SAX</td>
<td>25</td>
</tr>
<tr>
<td>3.8 Existing XML validation tools</td>
<td>25</td>
</tr>
<tr>
<td>3.8.1 Relax NG</td>
<td>25</td>
</tr>
<tr>
<td>3.8.2 Schematron</td>
<td>26</td>
</tr>
<tr>
<td>3.9 What existing tools lack</td>
<td>27</td>
</tr>
<tr>
<td>3.9.1 Schematron</td>
<td>27</td>
</tr>
<tr>
<td>3.9.2 Relax NG</td>
<td>27</td>
</tr>
<tr>
<td>3.9.3 XSLT</td>
<td>27</td>
</tr>
<tr>
<td>3.10 Literature review</td>
<td>27</td>
</tr>
<tr>
<td>3.11 Usability</td>
<td>28</td>
</tr>
<tr>
<td>4 The language</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Introduction example</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Structure validation</td>
<td>30</td>
</tr>
<tr>
<td>4.3 Statements</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Solving the test case</td>
<td>31</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>33</td>
</tr>
<tr>
<td>5.1 Reading XML</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Processing XML</td>
<td>34</td>
</tr>
<tr>
<td>5.3 Validating structure</td>
<td>34</td>
</tr>
<tr>
<td>5.4 Scanner</td>
<td>34</td>
</tr>
<tr>
<td>5.5 Abstract grammar</td>
<td>35</td>
</tr>
<tr>
<td>5.6 Parser</td>
<td>36</td>
</tr>
<tr>
<td>5.7 Implementation design</td>
<td>36</td>
</tr>
<tr>
<td>5.7.1 Generating Code</td>
<td>36</td>
</tr>
<tr>
<td>5.7.2 Arithmetic</td>
<td>38</td>
</tr>
<tr>
<td>5.7.3 Name analysis</td>
<td>38</td>
</tr>
<tr>
<td>5.7.4 Predefined functions</td>
<td>38</td>
</tr>
<tr>
<td>5.7.5 Collecting Error</td>
<td>39</td>
</tr>
<tr>
<td>5.7.6 Handling null references</td>
<td>39</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>41</td>
</tr>
<tr>
<td>6.1 Testing</td>
<td>41</td>
</tr>
<tr>
<td>6.1.1 The interview</td>
<td>42</td>
</tr>
<tr>
<td>6.1.2 Results</td>
<td>43</td>
</tr>
<tr>
<td>6.1.3 Discussion</td>
<td>43</td>
</tr>
<tr>
<td>6.2 Comparing with Java</td>
<td>43</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>45</td>
</tr>
<tr>
<td>7.1 Useful resources</td>
<td>45</td>
</tr>
<tr>
<td>7.1.1 Lack of tools</td>
<td>46</td>
</tr>
<tr>
<td>7.2 Implementation</td>
<td>46</td>
</tr>
<tr>
<td>7.3 Testing</td>
<td>46</td>
</tr>
<tr>
<td>7.4 Future work</td>
<td>46</td>
</tr>
<tr>
<td>7.4.1 Additional types</td>
<td>47</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 Motivation

When information is shared between networks it might be important to check that the information sent is correct and that no sensitive information is being sent by mistake or malicious intent. Today this can be solved with different kinds of validation solutions [18][8][26][14]. Validation means checking that the information match some requirements set for being used. In this thesis the way information is used is by sharing.

The validation could be a hardware solution, meaning that the hardware only allows certain information being sent. It could be a human checking the information by hand. Or it could be software processing the information before sharing it.

An example situation might be on a hospital. A doctor that want to share journal information about a patient but only want to share relevant information and leave out sensitive patient information, see Figure 1.1. The doctor might know what is allowed to send, but might not have time to edit out sensitive parts or simply forget to do it. Manual checking of all journals requires a lot of resources, thus automatic checking of them is desirable.

Hardware validation solutions are static to the system they apply too. This means that the hardware needs to be changed when the system changes, and changing hardware is hard work. Humans can deal with changing requirements and perform new kinds of validation on information, but having a human check information includes the risk of human mistakes. And if large pieces of information is shared it would require a lot of work. It would be useful to have a software validation solution since software makes less mistakes and works fast.

Information is a very broad concept. To be able to validate information the solution has to know what interchange format the information will be in. In this thesis XML was chosen, primarily because there are many tools for working with XML [19][6].

This master’s thesis was carried out with Advenica. What Advenica sought to do was to create a software validation solution. This solution should be able, with ease, to adapt
Figure 1.1: A doctor want to send patient journal information to a colleague

according to requirements. To achieve this Advenica had an idea of creating a language. With the language rules should be written, now called validation rules. Validation rules are the requirements that are required for an XML document to be used. A set of validation rules will be called a program. A validator would be generated from a program. A validator takes information as input and validate the information according to the validation rules written with the language. The validation rules should be able to use complex computations for the given input to validate, e.g. compute distance from two coordinates on a sphere and validate if the distance is too long or short. If the validation pass then the information can be used safely.

The validation rules must in some cases be verified by human eyes, verifying the validation rules means confirming that the rules match the requirement set upon them. If one writes validation rules for exporting sensitive information from one closed network to another it would be useful, might even be required by policy, for the validation rules to be externally verified by a third party before being put into use. The consequences of human mistakes in such automated systems could potentially exceed those with manual labor by a wide margin. Thus the validation rules has to be readable to aid in the verification of the validation rules. Readable in this thesis means that the person reading the validation rules should with ease be able to understand the syntax, functionality, and purpose of the validation rules. The person verifying might not have a level of experience with programming equal to the programmer whom created the validation rules.

### 1.1.1 Contribution

This thesis work resulted in a language which can define validation rules for XML documents, which will now be referred to as CheckXML since the rules check the XML, according to Advenica’s requirements. CheckXML was evaluated with usability tests to confirm and improve the readability.
1.2 Methodology

This section describes the process of making CheckXML. Each subsection will introduce one research question (RQ), which will be answered and/or analysed in this thesis.

1.2.1 Useful resources

The process of creating CheckXML begins with investigating useful resources. There are languages and tools that validate XML e.g. [18][26], this investigation aims to find out if they can be used in the implementation.

- RQ1: What useful tools exist and how can they be used to create CheckXML?

There are a lot of useful tools that process XML. This investigation will be limited to the most known and used tools.

1.2.2 Creating a validation language

Advenica provides a test case and with requirements on how the test case should be solved. CheckXML will be developed with an aim of solving the test case.

The compiler for CheckXML will be generated using some tool. More details on how the compiler is generated and what external tools is described in Section 2.2. This RQ is a practical question rather than a research one.

- RQ2: How to design and implement CheckXML?

1.2.3 Testing readability

Readability is not easy to measure in a language. In this case it is important to make CheckXML readable for non-programmers. The aim will be to first make a language and then test it with usability testing and take the feedback from the usability testing and improve CheckXML.

- RQ3: How to make CheckXML readable for non-programmers?

The usability testing will be an interview where the persons interviewed will try to verify some validation rules for a test case.
1. Introduction
This chapter will further define the concept of a validator and how validators will be generated in this case. The test case from Advenica will also be presented.

### 2.1 Validator definition

In the Introduction a validator was defined as a tool that takes information and validates the information according to validation rules. However, the formal definition is that a validator should:

- **Confirm well-formness. [9]**
  This checks that the syntax of the XML is correct. The syntax of an XML document is what makes the document XML, so it is arguable whether this belongs to the validator or belongs to the definition of XML. It will be clear how an XML document will look like later in Section 3.2.1.

- **Confirm validity. [9]**
  This checks further constraints set upon the XML by some rules. This could be structure or data type constraints on the XML document. Further analysis of this in Section 3.4.

### 2.2 Validator construction

*CheckXML* contains validation rules. In order to generate a validator from the validation rules there is a need of a *compiler*. A compiler transform source code (in this case validation rules) written in a programming language (in this case *CheckXML*) into another language (in this case Java).
The development of CheckXML revolved around the test case. The test case was provided by Advenica and consists of two parts:

1. An XML document containing coordinates and timestamps of a route from Ängelholm (SourceLocation) to Lund (Destination) through three cities (Waypoints).

2. Restrictions on how the coordinates can be formed and a maximum speed between the waypoints.

Reasons for choosing XML is discussed in Section 3.3.

There are three types of coordinates in the test case. SourceLocation, Destination, and Waypoint. They all have the same attributes: Lat, Long, Alt, and TimeStamp.
The complete test data is in A.1.
The restrictions are:

- $-90 < \text{Lat} < 90$
- $-180 < \text{Long} < 180$
- $-10000 < \text{Alt} < 10000$
- Maximum speed between Waypoints is 120 km/h.

2.3.1 Computing the distance between coordinates

There are several ways to calculate the distance between two coordinates on a sphere. The one used in this test case is haversine. The distance is calculated as:

$$d = 2 \cdot r \cdot \arcsin(\sqrt{\text{haversine}(\phi_2 - \phi_1) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \text{haversine}(\lambda_2 - \lambda_1)})$$  \hspace{1cm} (2.1)$$

The $d$ is the distance between the two coordinates which has latitude $\phi$ and longitude $\lambda$. $r$ is the radius of the earth, in this case 6371 m. [? ]

If the formula is to be used in the solution it is required to compare coordinates from different elements with each other.
Chapter 3

Background

This chapter covers background on what type of compiler is constructed for *CheckXML*, what tools are used for working with data, related work, and usability testing.

3.1 Compiler

This section will describe what tools are used to construct the compiler for *CheckXML* (use Figure 2.1 for reference), how they work in general, and a few examples. The input to the compiler is the validation rules and the output is a validator.

3.1.1 JastAdd

JastAdd is a tool for generating language-based tools. In this thesis it is used to generate the compiler for *CheckXML*.

JastAdd is a combination of reference attribute grammar and object-orientation. Context-free grammar describes the rules of a grammar that can be applied to any context. Attribute grammar extends context-free grammar with attributes. Reference attribute grammar is an extension to attribute grammar, which allows attribute grammars to have references. The abstract syntax tree is a tree consisting of nodes which represents the structure of the source code written in some programming language. JastAdd allows attributes to be defined on the abstract syntax tree. [16]

There are two kinds of *attributes* which can belong to a node in an abstract syntax tree: synthesized and inherited. These attributes gets their values from *equations*. Synthesized means that the attribute gets its value from an equation containing current or children attributes. Inherited means that the equation defining the attribute gets its value from an ancestor in the tree. [20, p. 130]

It is also possible to make the abstract syntax tree more extensive without using the parser. This is done using a *non-terminal attribute* (NTA). An NTA is similar to a node as
it can have attributes, but it is also like an attribute since it is defined and created by an equation. [15]

Attributes are declarative, which means that the attributes are not allowed to contain any side-effects. [15]

JastAdd has support for aspects orientation, which is a way of modularizing the code. In JastAdd, it is possible to have declarations that are defined outside the AST class they belong to, which are weaved into the class by JastAdd. In Chapter 5, aspects will be further described as they are used.

### 3.1.2 Scanner

Although JastAdd can do many things, scanning is not one of them. This section describes the scanner used with JastAdd to make the compiler.

A scanner is a tokenizer which recognizes patterns in the source code of the program and returns tokens. These tokens are then passed to the parser. A token is a string of characters specifically made for parsing. A pattern describes the form that the tokens can take [12, p. 111]. In CheckXML and the example Listing 3.1 regular expressions are used to form these patterns.

The scanner generator for the language in this thesis is the Java version of Flex, JFlex. Flex is a tool for generating scanners[3]. JFlex is a tool for generating scanners in Java [11]. JFlex takes a specification, which consists of macros and lexical rules, and generates a scanner. The macros are names for regular expressions. Lexicals are rules to specify tokens.

**Example**

The example Listing 3.1 defines one macro called Float on the first line. This macro matches a floating number which is zero or more numbers followed by a dot, followed by one or more numbers. The second line is a lexical rule which creates a token Terminals.FLOAT if a floating number is matched. Terminals.FLOAT is a value imported from the parser.

```
Listing 3.1: Simple JFlex example to recognize a float

Float = [0-9]*(\.)?[0-9]+;
{Float} { return Symbol(Terminals.FLOAT); }
```

### 3.1.3 Parser

In a compiler, the parser performs the syntactic analysis according to a formal grammar. There are two common strategies for parsing: top-down and bottom-up [12, p. 192] of the parse tree. In this thesis the bottom-up type of parser is used. Bottom-up starts from the bottom left end of a program with validation rules and works its way to the top, using rules from the formal grammar [12, p. 233].

For this thesis the tool Beaver is used to generate a parser for the language. Beaver generates LALR(1) parsers, it accepts EBNF, and works well with JFlex [1]. Beaver was chosen because it is fast, simple, works well with JFlex, and the generated parser is in Java.
Example

The Listing 3.2 is a beaver specification that is used to generate a parser. The parser accepts programs containing a sequence of floating numbers.

**Listing 3.2:** Simple Beaver example

```plaintext
%goal program;

program = stmt_list;

stmt_list = stmt | stmt_list stmt;

stmt = FLOAT;
```

If the beaver specification in Listing 3.2 would be used on the following program, using the scanner from Example 3.1:

```
4.15 65.12
```

the resulting parse tree is Figure 3.1.

### 3.1.4 Formal grammar

The formal grammar used for *CheckXML* is called context-free grammar. It consists of non-terminals, terminals, and productions.

A non-terminal is a symbol that is the result of a rule. A terminal symbol is a literal symbol which are used, with other non-terminals, to make the rules of the non-terminals. The grammar has tokens defined by the lexical rules as its terminal symbols. The production rules specify how the terminals and non-terminals are combined. [12, p. 197]

A production consists of a left-hand side and a right-hand side. The left-hand side contains the non-terminal. The right-hand side contains zero or more terminals and/or non-terminals, which describes how the non-terminal on the left-hand side is produced. [12, p. 197]
Example

There are several notations to express the formal grammar. The one used in this thesis is Extended Backus–Naur Form (EBNF). The grammar for Listing 3.2 is expressed in Listing 3.3. The grammar example describes that a Program can contain zero or more Statement's.

Listing 3.3: Grammar expressed in EBNF for float example

```plaintext
Program = Statement*;
```

3.2 Data structure

This section will describe advantages and disadvantages with different types of data interchange formats, like XML and JSON.

CheckXML should be able to handle all kinds of data, however, to make input uniform it has to have some structure. The act of giving data this structure is outside the scope of this thesis. Some sort of preprocessor is needed to transform data to some data structure before the validator from CheckXML can perform the validation.

3.2.1 XML

XML (eXtensible Mark-up Language) is a way of structuring data which is built to be easy to read, both by humans and programs [9]. XML uses tags to form elements which can contain more elements or values. Elements can have attributes. Attributes can only have a single value.

Advantages

- Was made to be straightforwardly used over the internet. [9]
- XML documents are easy to create. [9]
- Supports namespace. Which means that it is possible to enforce a certain set of rules to prevent element name conflict.
- There exist well developed schema and transformation languages.
- Java has library support for XML and related tools.

Disadvantages

- XML syntax is verbose. When a lot of data is being represented in XML it can be hard to get an overview of the data. The larger the document is the more descriptive the tag names have to be. Longer tag names make XML documents look clumsy.
- No support for data types.
• A lack of sufficient processing applications. For subsets of XML, like HTML, there are processors, like browsers, but for XML there are no standard.

3.2.2 JSON

JSON (JavaScript Object Notation) is another way to represent data. It is a language-independent data format which is formed of a subset of JavaScript. A JSON data interchange format consists of objects that have values. The value can be an object, array, string or number. [4]

Advantages

• It is lightweight, which makes it easy to read and get an overview of. [4]

• It is easily integrated in JavaScript. [4]

• Stores data in arrays and records, which makes it easy to work with in languages since there is no need to transform. [4]

Disadvantages

• Limited support for Java development.

• Mainly used for web development and not applied in other systems. [4]

• No namespace support. Meaning there are extension to validate namespace, manually prefixing would solve this. Though there are standalone extensions for this.

3.2.3 Example

Listing 3.4 is an example of a simple XML document which contains information, author, title, and price, about two books.

Listing 3.5 is the same information as Listing 3.4 but in JSON.
3.3 Choosing data format

While JSON is gaining popularity as a data interchange format on the web, XML is still dominant. Since CheckXML is supposed to support varying systems, back in Section 3.2 this was discussed, it would be preferable to use XML. [25]

XML has the benefit of being the universal standard and thus there are more tools to process XML, and more schema languages to validate XML. While JSON is easier to learn and simpler to use, XML was chosen to represent data in this thesis.

Thinking back to the introduction example in the Section 1.1 the patient journal could be transformed into an XML document and then used as input to the validator. The responsibility of transforming the journal to XML lies with the hospital.

3.4 XML validation

There are several tools available to validate XML, in this thesis the two main tools are discussed: XML Schema Definition (XSD) and Document Type Definition (DTD), these two are the most used validation tools used today. Both schemas work for validating XML, but they have differences:

- XSD is written in XML, while DTD has a unique syntax from SGML DTDs [14, ch. 5]
- XSD has typing [14, ch. 5 p. 5] while DTD has not. Typing means that content can have types.

Listing 3.4:
XML example

```xml
<books>
  <book id="001">
    <author>Gambardella</author>
    <title>XML Guide</title>
    <price>98.50</price>
  </book>
  <book id="002">
    <author>Ralls</author>
    <title>Midnight</title>
    <price>55.50</price>
  </book>
</books>
```

Listing 3.5:
JSON example

```json
{
  "books": [
    {
      "id": "001",
      "author": "Gambardella",
      "title": "XML Guide",
      "price": "98.50"
    },
    {
      "id": "002",
      "author": "Ralls",
      "title": "Midnight",
      "price": "55.50"
    }
  ]
}
```
• XSD has occurrence constraint [14] while DTD has not. Occurrence constraint are rules defining frequency of a certain element.

It is important to note that XSD and DTD can not make comparisons with content/attributes and other content/attributes. These validators only validate the syntax and set a few constraints upon the content of XML documents. This will be further discussed in Section 2.3 and Section 3.9.

XSD will be used in CheckXML.

3.4.1 Example

Checking the well-formness of XML would be to check the following:

• Content of elements is defined.

• Content is within a beginning and end tag.

• Content is properly nested; there are parents within roots and children within parents.

Checking the validity of XML, in this case with XSD, would be to check the following:

[14]

• Element declaration.

• Attribute declaration.

• Simple and complex types.

There are more things that can be checked with XSD, but these are the ones used in this thesis.

3.5 XPath

XPath is a way of selecting certain nodes in an XML document. It is designed to be used by e.g. XSLT which is an XML document transformer. In addition to selecting parts of an XML document, XPath has support for basic manipulation of strings, numbers and booleans. XPath uses four types: node-set, boolean, number, and string. [6]

To retrieve the nodes XPath uses expressions, or queries. A query consists of a path to the desired nodes. To pick the first book in Listing 3.4 the expression would look like this: /books/book[0] and would return a Node object with the attributes from the first book.

XPath has predefined functions which can be used to determine more specific information about the selected nodes or to improve the selection of nodes. The functions are embedded in the expression. Here follows a few example of functions, all examples use the XML from Listing 3.4:

• position(), returns the position of a node. [6]
3.6 XSLT

XSLT (Extensible Stylesheet Language Transformations) is a language to transform XML documents to other XML documents. It follows the same syntax as XML. XSLT uses XPath to navigate in the source XML document. [8]

Example

Listing 3.6 picks the authors from the books in the XML Example 3.4 and outputs an XML file with the title and authors of all books.

Listing 3.6: Simple XSLT example

```xml
<?xml version='1.0'?>
<xs:stylesheet version="1.0"
    xmlns:xs="http://www.w3.org/1999/XSL/Transform">

    <xs:output method="xml" omit-xml-declaration="yes"/>

    <xs:template match="/"
        <xs:for-each select="books/book">
            <xs:apply-templates select="title"/>
            <xs:apply-templates select="author"/>
            <BR/>
        </xs:for-each>
    </xs:template>
</xs:stylesheet>
```

Important to note here is that XPath is used to match the values from the XML.

3.7 Parsing XML

This section is about traversing XML documents. There are mainly these two ways of doing this: DOM and SAX.
3.7.1 DOM

DOM (Document Object Model) is a platform- and language which parses the entire XML document and stores the document in memory as a tree, which can then be accessed by the user. [2]

3.7.2 SAX

SAX (Simple API for XML) parses the XML and uses callbacks during the parsing which are called upon when events occur. SAX only looks at one element at a time and does not store complete XML document in memory. This means it is more memory efficient than DOM since the max amount of memory needed is the maximum depth of the node structure. [5]

3.8 Existing XML validation tools

There exists a set of tools called Document Schema Definition Languages (DSDL). DSDL works with validation of XML documents with the goal of a more advanced validation than that of a single tool, which is what this thesis is trying to achieve, see Section 1.2 RQ1. This section consists of a short investigation of two of these: Relax NG and Schematron.

3.8.1 Relax NG

REgular LAnguage for XML Next Generation (Relax NG) is a regular-grammar based language for XML. A Relax NG schema is written in XML and is similar to XSD, but was made to be simpler. Relax NG was developed by the same time as XSD but was not as popular since XSD became W3C recommendation. [7]

Example

The example Listing 3.7 validates the XML in Listing 3.4.

**Listing 3.7:** Simple Schematron example

```xml
<element name="books"
    xmlns="http://relaxng.org/ns/structure/1.0">
    <oneOrMore>
        <element name="book">
            <element name="author">
                <text/>
            </element>
            <element name="title">
                <text/>
            </element>
            <element name="price">
                <text>25</text>
            </element>
        </element>
    </oneOrMore>
</element>
```
The root node is books and it can contain one or more book and each book has three attributes with text.

3.8.2 Schematron

Schematron is a rule-based language for finding pattern for existence or lack of existence of nodes in XML trees [18]. Schematron is based on XSLT and is similar to XML Schema and DTD but with more features:

- Let a value of an element depend on its siblings [18], which means that the content can be controlled by one of its siblings.

- Set rules for relations between XML files. [18]

Schematron takes two steps in order to validate the rules. It finds the desired content using XPath 3.5. Then it checks to see if the XPath expression is valid.

There are two kinds of schemas the are used in Schematron. Report and assert. Report diagnoses which variation of language is dealt with. Assert confirms a particular schema. [18]

Schematron is the most successful language in dependency, however it is worth mentioning that a similar language, SchemaPath, which extends XSD with a dependency implementation. SchemaPath is useful since it is not a completely new language, like Schematron. [26]

Example

Listing 3.8 is an example of a Schematron validator that will test if any book has a title longer than 100 characters from Listing 3.4.

Listing 3.8: Simple Schematron example

```xml
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern>
    <title>Price rules</title>
    <rule context="books/book/title">
      <assert
        test="string-length(normalize-space(.)) &lt; 100">
        Title too long.
      </assert>
    </rule>
  </pattern>
</schema>
```
3.9 What existing tools lack

The idea with the language is to use the useful parts of tools and add extensions, *CheckXML*, to do what the tools are lacking.

In Section 3.4, 3.5, 3.6, and 3.8.2 tools for working with XML are presented. What these tools lack is a kind of computational dependency between elements. In the test case the speed between two waypoints has to be calculated and evaluated. This means that the validation of the XML document relies on a waypoint’s relation to another waypoint. This relation requires complex computations. There is currently no validator to check the computational dependency between two elements so it has to be done with *CheckXML*.

3.9.1 Schematron

The test case could not be solved using Schematron. Schematron can set rules for sibling nodes, as mentioned in Section 3.8.2, but there is no support for mathematical operations. It will be clear why this is needed in Section 2.3.

Even if Schematron can use siblings to control content it is not possible to control arbitrary elements with each other.

3.9.2 Relax NG

The test case could not be solved using Relax NG. Relax NG works in a similar fashion to XSD but with a simpler syntax, this makes Relax NG a more suitable solution to match readability.

3.9.3 XSLT

XSLT is a transformation tool. Meaning the test case XML document would have to be transformed into something. This something would be the result of a kind of validation.

If an XSLT program contains a set of rules and outputs only if the rules are matched it is possible to check the output file if and what it contains and draw conclusions about the input XML document, or create a safe XML document.

There are two reasons for not using XSLT in *CheckXML*:

- XSLT does not have dependency among nodes.
- XSLT is a “right tool for the right job” kind of tool. Meaning it is good when you specifically need to transform XML, but not otherwise.

3.10 Literature review

One major drawback with current major XML validators, DTD and XSD, is that

*Definition of an element cannot depend on its context.* [27][23, p. 112]
Aside from the major validators there are many recent validation languages that attempts to remedy this specific drawback, e.g. Schematron implements conditional definition \[22\] and DSD implements rules that can depend on elements of the current context \[23, p. 159\]. The motivation why this drawback is crucial is described in Section 2.3.

This thesis follows the same spirit as the paper \textit{SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints} [26] which implemented conditional constraints for XSD, but with more focus on readability and mathematica operations.

Schematron and SchemaPath solutions solve the dependency issue, however they do not solve some computational issues presented in the Introduction.

XSD’s are often viewed as large and complex [19] and does not focus on being readable in the sense describe in Section 1.1, even though XSD has XML syntax and XML is supposed to be human-legible [9].

### 3.11 Usability

Usability is usually of narrow concern compared to whether the system is good enough to satisfy all the requirements. Even the term usability is vaguely defined. It can be said to apply to all aspects of a system with which a human might interact, in this case being \textit{CheckXML}. Since usability is not a single property it is traditionally divided into these five attributes: [24, p. 25]

- Learnability: the language should be easy to learn so that the user can start verifying it as soon as possible.

- Efficiency: the language should be efficient to use. Which essentially means that a user should be able to efficiently be able to determine the correctness of the validation rules.

- Memorability: the language should be easy to return to after some period of not having seen it, without having to learn it all over again.

- Errors: the language should have low error rate, so the user does not make errors.

- Satisfaction: the language should be pleasant to use.

When testing the usability of \textit{CheckXML} the primary aspect is to investigate how the user interpreters the validation rules. This has to do with the attributes learnability and efficiency.

Learnability since the user has to learn and understand what the validation rules are doing and make the connection to the requirements on the XML.

Efficiency since the user cannot spend a large amount of time trying to understand a certain validation rule. Each rule should be simple and intuitive.

Jakob Nielsen mention that: \textit{Thinking aloud may be the single most valuable usability engineering method} [24, p. 195], this method is also advocated in [21]. These are the reasons the "think aloud" method is used in this case. The aim is to understand what the persons interpreting a program are thinking.
This chapter describes the language, CheckXML, that resulted from the implementation and testing. The chapter consists of examples, tools for working with data, description of the functionalities of CheckXML, and lastly how the test case is solved using CheckXML.

CheckXML contains a structure and computational validation. The structure validation is specified using XSD and the computational validation consists of validation rules. CheckXML generates validators. Running the compiler on a program written with CheckXML will output an .XSD file and .java file. The XSD file checks the structure of the XML and the Java file takes an XML document as input and validates it using the rules in the program.

4.1 Introduction example

Listing 4.1 serves as an introduction to the computational validation of CheckXML. The first line declares a variable costAllBooks which contains a number from the XPath expression on the right-hand side. The expression returns the sum of all prices of all books from Listing 3.4.

The second line is a rule for checking if the variable costAllBooks is lower than 500. If the boolean expression is True then the validation on line 2 will pass.

The forth line begins an iteration over all the books. Each iteration declares a variable price which is the price of the book in the current iteration. price is then used in a CHECK to see if the price of the current book is below 100.

Listing 4.1: Introduction CheckXML example

costAllBooks = #sum(/books/book/price)#
CHECK costAllBooks < 500
FORALL book IN #/books/book# {  
   price = book#price#  
   CHECK price < 100  
}

4.2 Structure validation

It is obvious, from Section 3.4, that good support for XML validation exists, and that there are two main tools for it. XSD got bigger and was published as a W3C recommendation in 2001 [10]. This motivates the use of XSD in CheckXML instead of DTD.

4.3 Statements

This section will cover CheckXML statements in detail. There are five kind of statements. Each will be described and explained with an example.

- XSD. It is the first part of a program and is the same as ordinary XSD.

- Variable declaration. Variables are defined either with an XPath query to the location in the XML enclosed with hash tags or with an arithmetic expression. Examples:
  
a = 5  
b = #/books/book[1]/price#

  The variable a is a number, this is obvious. The variable b is also a number, this is less obvious. All variables in CheckXML are numbers or an expression to extract a number from the XML document.

  If a variable is declared with an XPath expression and the XPath expression is invalid (returns nothing) an exception will be thrown.

- Check. The Check contains a boolean expression, and if the expressions of all check statements are evaluated to True, the validation is successful. All Check statements that evaluate to false will be reported as error messages containing the variables involved.

  CHECK a < b

  If the price of the first book is less than 5 this will output an error containing the variables a and b.

- Function. Creating and calling a function with Double variables. Here the variable c is being declared with the function add. Calling add(3,4) returns 7.

  DEFINITION add(a, b) {
      return a + b
  }

  c = add(3,4)
There are predefined functions as well, for example: \( \sin(\text{Double}) \), \( \cos(\text{Double}) \), \( \sqrt{\text{Double}} \), and \( \arcsin(\text{Double}) \).

- **FORALL** \(<\text{single}>\) **IN** \(<\text{set}>\). This is an iteration that iterates over all elements in \(<\text{set}>\) with the reference element \(<\text{single}>\). The ID used in \(<\text{single}>\) can be used to extract content from each element in the \(<\text{set}>\). This is exemplified in Listing 4.1.

To solve the test case a WayPoint has to be compared with the next (or previous) WayPoint. In the body of the **FORALL** it is possible to use a notation \( \text{PREV} \) before an XPath expression. Using the notation \( \text{PREV} \) on an expression takes the node before the current one. But then there is the issue of the first node, which does not have a previous node. This was solved by implementing an **EXCEPTFIRST** which should be written after the \(<\text{set}>\) in the **FORALL**.

### 4.4 Solving the test case

By looking at the test data in A.1 we notice that there is no difference between the Source-Location, the Destination and the WayPoints element. Thus they can be transformed into WayPoint elements. This allows for a solution using iteration. Using iteration makes the solution scalable, in case more WayPoint’s are added or removed.

The full program is in A.4. This version is after the testing was done. Earlier version will be discussed in Section 6.1.

The restriction of the coordinates is validated with XSD. XSD has restriction constraints which can be set for each node. For the coordinate node the Lat, Long, and Alt are restricted with \( \text{minValue} \) and \( \text{maxValue} \) according to the requirements from the test case.

For the computational validating an iterator is used with the **EXCEPTFIRST** notation. On each iteration a variable is declared and initialized for the Lat and Long for the current and previous element. The values are then sent as arguments to a function which computes the distance between two coordinates on the earth using haversine. Haversine uses predefined mathematical functions to perform the computations. The time is extracted and reformed to hours. Then the speed check is performed with a CHECK to make sure the speed is below 120 km/h. In Listing 4.2 the computational validation of the solution is presented.

#### Listing 4.2: Validation rules for the solution for the test case

```java
DEFINITION haversineFunction(diff) {
    haversine = (1 - cos(diff)) / 2
    RETURN haversine
}

DEFINITION distanceCoords(long1, lat1, long2, lat2) {
    squareRoot = sqrt( haversineFunction(lat2-lat1) + 
    cos(lat1)*cos(lat2)*
    haversineFunction(long2-long1) )
    result = 2 * 6371 * arcsin(squareRoot)
}
```
RETURN result
}

FORALL coordinate IN #/Message/Coordinates# EXCEPTFIRST
  prevLat = PREV coordinate#Lat#
  prevLong = PREV coordinate#Long#

  currLat = coordinate#Lat#
  currLong = coordinate#Long#

  prevTime = PREV coordinate#Time#
  currTime = coordinate#Time#

  distance = distanceCoords(prevLong, prevLat, currLong, currLat)

  time = currTime - prevTime
  timeInHours = time / 60 / 60
  speed = distance / timeInHours

  CHECK speed <= 120
}

The result of compiling the program is a validator in a .java file. The validator is then run with the XML as input and gives the following output:

Failed CHECK speed(164.9972653810656) <= 120(120) on iteration 3

So there was one speed that was above the speed limit. This means that the 3rd speed check failed. Meaning that the speed from Kristianstad to Landskrona was too high. Which agrees with what Advenica expected. To make the XML document pass the user would have to either remove the wayPoint Kristianstad or increase the time between the cities so that the speed is reduced.
Chapter 5
Implementation

This chapter describes the implementation of the CheckXML compiler.

The implementation followed the programming paradigm aspect-oriented programming (AOP). AOP is a way of modularising the code. There is a main structure which can be built on by adding code to aspects. The code in the aspects adds extensions to the main structure. In JastAdd these aspects have the format .jrag for declarative aspects. In CheckXML there are seven aspects:

1. ErrorHandling: dealing with errors during parsing.
2. NameAnalysis. Finding declarations for ID uses.
3. FunctionNameAnalysis. Finding function declarations for function calls.
6. UnknownHandling. Implementing null object pattern.

The full code for each aspect will not be presented in this report. However the design of the implementation is discussed in Section 5.7.

There exists useful tools for validating XML, one will be integrated into CheckXML. Since the there are not specific language or tool for the computation rules it will be created.

5.1 Reading XML

To read XML in Java the package org.w3c.dom.Document is used. A Document is an object that represent the entire XML document. To create the Document a Document factory is required. The factory does more than create Document objects, which will be further discussed in Section 5.3.
5.2 Processing XML

Java contains several methods for processing XML e.g. with the class
org.w3c.dom.traversal or org.xml.sax.XMLReader. These classes allows
access to the nodes by iterating over the tree (org.w3c.dom) or set event handling on
each node (org.xml.sax). These two classes allows for parsing of the XML, explained
in Section 3.7. The purpose is to process the XML to extract desired content from the
XML document. Using DOM or SAX would require additional methods for traversing the
XML document using the packages available. However, there is a simpler way to do it,
using XPath.

XPath allows for immediate selection of content from elements in XML documents us-
ing querying and it is recommended by W3C [6], thus it will be used in CheckXML. XPath
is simple to understand if the XML document is visualised as a tree, the XPath query looks
like a path to a document. XPath is supported in Java using the class javax.xml.xpath.

The query is a String with XPath syntax. The String is evaluated on the Docu-
m ent with a type constant. The constant is a way for the user to decide before evaluating
the query what type will be returned. XPath allows four types: BOOLEAN, STRING,
NUMBER, NODESET, and NODE.

5.3 Validating structure

The compiler reads the XSD lines and outputs a .xsd file called schema.xsd which con-
tains all XSD. In Java the XSD Schema is applied before the Document, see Section
3.3, is created. The factory that creates documents can have a fixed Schema set to it. A
Schema object can contain an XSD file which can validate the XML document. If the
XML document is invalid the generated validator will output error messages informing
the user what went wrong.

5.4 Scanner

In CheckXML the following regular expressions are used in the scanner:

- WHITESPACE = [ ] | \t | \f | \n | \r
- NUMBER = [0-9]*(\.)?[0-9]+   
- COMPARE = != | == | <= | >= | < | >   
- ID = [a-zA-Z][a-zA-Z0-9]*   
- PATH = #(.+?)#   
- SCHEMALINE = <(.+?)>

The following keywords and operators are added: DEFINITION, RETURN, CHECK,
FORALL, IN, EXCEPTFIRST, PREV, +, -, *, /, %, and =.
There is also an error fall-back in case none of the regular expression patterns above matched. If the error fall-back is matched an error will output to warn the user what pattern was read so that the user might identify what went wrong. The error fall-back has the regular expression which matches to anything.

The lexical rules are ordered. For each symbol the scanner tries to match a pattern. Trying the first regular expression pattern in the list above. This allows all the white space, new lines, etc to be matched before any other symbol. The error fall-back pattern is placed last to match everything that is not in the list.

## 5.5 Abstract grammar

An abstract grammar contains abstract syntax tree declarations which describes the parsing grammar of a language. This section contains the full abstract grammar for CheckXML.

The grammar of CheckXML is seen in Listing 5.1. The abstract classes are:

- **Stmt.** Statements are what makes a program, like sentences in a natural language. Statements make the sequence of actions that a program consists of, by using variables, functions and checks.

- **Data.** The Data is the base container for variables. The base container is used in arithmetic, as argument for functions and as arguments for check.

- **Arithmetic.** Uses Data elements to perform computations.

### Listing 5.1: Full EBNF grammar for CheckXML

```plaintext
Program ::= SchemaLine* Func* Stmt*;
SchemaLine ::= <SCHEMALINE>;

Func ::= FuncDecl Decl* Stmt*;
abstract Stmt;
Decl : Stmt ::= IdDecl Init;
Assign : Stmt ::= IdUse Data;
Check : Stmt ::= Left : Data Compare Right : Data;
Return : Stmt ::= Data;

Init ::= Data;
Compare ::= <COMPARE>;

abstract Data;
Number : Data ::= <NUMBER>;
NodeSet : Data ::= Path;
IdUse : Data ::= <ID>;
FuncCall : Data ::= IdUse Data*;
NoData : Data;
```
abstract Arithmetic : Data ::= Left : Data Right : Data;
Add : Arithmetic;
Sub : Arithmetic;
Mul : Arithmetic;
Div : Arithmetic;
Mod : Arithmetic;

IdDecl ::= <ID>;
FuncDecl ::= <ID>;
Path ::= <PATH>;

UnknownVariable : IdDecl;
UnknownFunc : IdDecl;

5.6 Parser

The parser begins with defining all the terminals. These are the terminals imported in the scanner, Section 5.4.

The abstract classes in the abstract grammar are the non-terminals that have several alternatives. In the parser the | sign is used to separate the alternatives.

Every list, <listname>_list_eps, is constructed in the same fashion, here using function as example:

function_list_eps = /* epsilon */ | function_list ;
function_list = function | function_list function ;

This example allows zero or many functions.

%left and %right gives the operator associativity of the arithmetic. In this case ordinary order is desired. So MUL, DIV, and MOD goes first.

The full parser can be viewed in Section A.2.

5.7 Implementation design

It is useful to look back at Section 5.5 for a reference on how a program is being structured and what node is currently being referenced in a section.

5.7.1 Generating Code

The implementation for the main code generation functionalities will be presented in this section. Listing 5.2 is a simple CheckXML example which generates the Java code in Listing 5.3. The variables expression1, XPath, and doc are defined earlier in the generated code. In this case there will be no error output since the price of the book is below 100.

Listing 5.2: Simple CheckXML example

CHECK #/books/book[@id='001']# < 100
Listing 5.3: Generated Java code from the CheckXML in Listing 5.2

expression1 = "/books/book[@id='001']";

double tempVar0A =
    (double) XPath.compile(expression1).
        evaluate(doc,XPathConstants.NUMBER);
double tempVar0B = 100;
if(!(tempVar0A < tempVar0B)) {
    System.err.println("Error at line 1: Failed CHECK \
"/books/book[@id='001']/\ +
(\"+String.valueOf(tempVar0A)\") < +
100(\"+String.valueOf(tempVar0B)\")");
}

The following subsections contains a description on how code is generated for the specific node.

Program

The Program node is the root of the AST. This will be the start of the traversal of the source code, which means that all the necessary files needed for the code generation can be defined here. This includes:

- Importing all packages necessary.
  This includes packages for XML, XSD, XPath, and exceptions.

- Declare and initialize all the necessary variables.
  This includes factories, schemas, xpath objects, and temporary strings.

- A request to generate code for all the children of the program node.

- Catch all exceptions.

- Define all predefined functions.
  This includes sin, cos, sqrt, and arcsin.

Declaration and initialization

When a variable is declared with an XPath query it should be transformed into a variable with the same name but with the extracted value from the XML. It is possible to declare variables with XPath queries and numeric expressions but in the generated code all variables are double. The XPath queries are assumed to return a number, if a query returns nothing then an exception is thrown with the bad query.

Variables in CheckXML are only allowed to be defined once.
Check

The Check node has two arguments. These has to be interpreted and compared according to the rule. This is done with the same method as in declaration.

The error message should contain the line of the failure, the variables involved and the number for each variable. And if the Check is within an iteration it should also print which iteration failed.

ForAll

The ForAll node has a base query in <set> which contains all the elements to be iterated over. The set is initialized and iterated over using index in XPath to perform the iteration.

A previous function is required to reach the previous waypoint in the test case. This means that the index before the current index in the XPath query.

5.7.2 Arithmetic

Since the test case requires mathematics to be performed, arithmetic expressions was implemented. Arithmetic works as in most languages. A variable can be assigned a mathematical expression of variables/function calls and operators, e.g. \(4 + a \times \text{sumOf}(b, c)\), with the operand evaluation order: multiplication, division, and modular, then addition and substraction.

5.7.3 Name analysis

The name analysis is implemented using an attribute lookup in accordance with usual techniques for name analysis using reference attribute grammars [15].

Multiply declared variables

By using the lookup attribute in the name analysis, an attribute isMultiplyDeclared is declared for the IdDecl node. It does the same thing as in the IdUse node but looks for a variable with the same name as the declaration node. If there exist such a node the value of the attribute will be true, and thus the name is multiple declared.

Function name analysis

The name analysis for functions works in a similar fashion as the name analysis for variables. The FuncCall node has an attribute funcLookup which looks for the declaration of the function that is called.

5.7.4 Predefined functions

To handle predefined functions in the name analysis for functions, an NTA is created that contains all the predefined functions.
The NTA is a node with attributes but is defined by an equation, in this case the equation defines the predefined functions. This NTA is present everywhere and thus can be called on to check for predefined functions. NTA’s are described further back in Section 3.1.1.

5.7.5 Collecting Error

JastAdd supports collection attributes. These attributes have composite values defined by contributions. In the CheckXML compiler, errors are represented as a collection attribute. For each kind of error, a contribution is added to this collection attribute. The contributions are:

- Undefined variables, functions, and function calls.
- Multiple variable and function declaration.

XPath queries are not yet supported in the current error handling, however, they will generate runtime errors. A solution to this is discussed in Section 7.4.2.

5.7.6 Handling null references

Invoking methods with null reference is illegal and results in runtime error. If the lookup function does not find any reference it would return null. To solve this a NTA is introduced, called Unknown. This attribute is used to define null objects. Now instead of invoking methods with null the Unknown used. This technique is called null object pattern [17].
Chapter 6
Evaluation

The evaluation of CheckXML consists of usability testing and comparing with a program written in (not generated) Java code.

It would be useful to evaluate CheckXML in a realistic test environment to check if the language meets its requirements defined in research question regarding readability (RQ3), Section 1.2.3. Since CheckXML is generating Java code it is of interest to compare CheckXML with code written directly in Java.

6.1 Testing

One goal of CheckXML was to make the programs readable for people without programming background. To check if CheckXML fulfilled these requirements several interviews was carried out. The persons interviewed, from now on called test subjects, had some basic knowledge of XML and computers, with background in IT administration, web design and hardware design. The test subjects had little to none programming knowledge.

The task given to the test subjects was to verify that the validation rules match the requirements. The result of the interviews are used to improve the readability of CheckXML. During the test the XSD part was excluded. This is because even if the testing would give some feedback on how to form the XSD language syntax it would be troublesome to make the changes. In Section 7.4 an alternative to XSD is discussed.

The program to be verified can be viewed in Section A.3. The XML input and rules on the XML was from the test case in Section 2.3. The test subjects did not have any knowledge about XPath or Haversine so these were explained briefly before the interview. A snippet of the program in Section A.3 can be viewed in Listing 6.1.

Listing 6.1: Part of the program used for testing

```java
FUNCTION DOUBLE haversineFunction(DOUBLE diff) {
    DOUBLE haversine = (1-cos(diff)) / 2
}
```
RETURN haversine
}

FUNCTION DOUBLE
distanceBetweenCoords(DOUBLE long1, DOUBLE lat1,
DOUBLE long2, DOUBLE lat2) {
DOUBLE earthRadius = 6371
DOUBLE squareRoot = sqrt(haversineFunction(lat2-lat1) +
  cos(lat1)*cos(lat2)*
haversineFunction(long2-long1))
DOUBLE result = 2 * earthRadius * arcsin(squareRoot)
RETURN result
}

DOUBLE sourceLat = #/Message/SourceLocation/Coordinates/Lat#
DOUBLE sourceLong = #/Message/SourceLocation/Coordinates/Long#
DOUBLE way1Lat = #/Message/WayPoints/WayPoint[1]/Coordinates/Lat#
DOUBLE way1Long = #/Message/WayPoints/WayPoint[1]/Coordinates/Long#
DOUBLE sourceTime = #/Message/SourceLocation/Time#
DOUBLE way1Time = #/Message/WayPoints/WayPoint[1]/Time#

DOUBLE distance = distanceBetweenCoords(sourceLong, sourceLat,
  way1Long, way1Lat)
DOUBLE time = way1Time - sourceTime
DOUBLE timeInHours = time / 60 / 60
DOUBLE speed = distance / timeInHours
CHECK speed <= 120

...

6.1.1 The interview

Each interview consisted of two persons: one interviewer and one test subject. The interview is divided into two parts: introduction and evaluation. The first part introduces the test subject to the example input XML, Listing A.1, and rules on how to validate it, Section 2.3. In the second part, the test subjects attempts to verify that the CheckXML program is correct and fulfils the requirement.

As mentioned in the Background: the "think aloud" method will be used. The test subject were told verify that the validation rules match the requirements and speak their mind about their understanding of it.
6.1.2 Results

The functions were easy to understand as they only contained a mathematical formula. But the call to the function was not always obvious with the parameters and an expected return value. The return value was hard to interpret in a few cases. The function notation was not intuitive for everyone except one of the test subjects. Most test subjects tried to make their own notation before moving on. Saying things like: definition, defines, describes.

Checking the longitude, latitude and altitude with CHECK was easy and intuitive for all test subjects. Everyone saw the check as some sort of checker and concluded that the program performed the checks according to the program and the requirements.

All five of the test subjects spent a lot of time trying to guess what DOUBLE meant. After a while they simply saw past it as something they did not understand.

Test subjects often had to spend some time in the first speed check, however after the first iteration of speed check was confirmed it was pretty straightforward that the rest of the speed checks did the same thing but for different coordinates.

Most test subjects were happy with the structure of the program and the naming of the variables. These two features seemed to aid a lot in the understanding.

6.1.3 Discussion

The results proves that it is important to use a good structure and variable naming for the program. This minimizes the users own interpretation of the program which makes it easier to understand. It is not obvious if this practice could be implemented into CheckXML so the programmer is forced to use good practice when making the programs. But it would certainly be beneficial.

Functions were not natural for the test subjects. A function call had the same name as the declaration in the beginning of the program but it was not clear that it returned the result. A more intuitive notation may allow for easier interpretation.

Since the iterations were simple to understand as long as the first iteration was understood it motivates for the use of an iterator. An iterator would only introduce the statements once, and since the test subject understood the first iteration it would seem logical to have an iteration technique in CheckXML.

Even if the language is imperative it can be interpreted as declarative since variables only can be declared once. The test case was probably interpreted with a declarative sense. But it would be interesting to make further interviews.

6.2 Comparing with Java

Since the generated code is Java it is possible to solve the test case using only Java. It is interesting to see how this solution performs compare to the CheckXML solution. The main performance issue looked upon in this case is the code size.

The size testing can be seen as one kind of complexity comparison. If the code size is small that means the complexity is small. The way this is compared is by comparing lines of code, excluding empty lines and comments. The tool used to count lines of codes is CLOC [13].
Table 6.1: A table containing the lines of code required for solving the validation of three test cases.

<table>
<thead>
<tr>
<th>Test case</th>
<th>CheckXML</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinates</td>
<td>21</td>
<td>84</td>
</tr>
<tr>
<td>Contacts</td>
<td>13</td>
<td>80</td>
</tr>
<tr>
<td>Books</td>
<td>15</td>
<td>100</td>
</tr>
</tbody>
</table>

Using CLOC on a .java file which was created with the purpose of doing what the program in CheckXML does but as simple as possible, without the redundancy from generating code. The result is presented in table 6.1. In addition to the solution there are two more test cases:

- The Contacts test case are validation rules that check that all ages in the contacts data match their respective birth year according to the current year.
  
  Example:
  An XML document contains information about contacts, e.g. current age and birth year. If the birth age and current age from a contact is summed and does not equal 2015 then there should be an error.

- The Books test case are validation rules that check the price (with tax) for all books are below 50, the sum of all prices from all books is below 1000, the sum of all books from a specific author is below 20, and that the total number of books is below 10.
  
  Example:
  An XML document contains information about books, e.g. author and price. If the sum of all prices from all books exceed 1000 then an error will be thrown.

The XSD was not included in this comparison since there would be no difference in the lines of code.
Chapter 7

Conclusion

In this thesis a language, CheckXML, is designed for validating XML documents. CheckXML consists of validation rules. A compiler is implemented which interprets the validation rules and generates a validator in Java, which contains the validation rules.

The language is tested regarding its usability. There were two tests performed: usability and size. The usability testing was done with a "think aloud" interview method with people whom verifies the validation rules. The size testing was done on three test cases validating different kinds of information.

CheckXML solves problems where contents from XML have to be compared to each other or numeric constants. There are only numbers in CheckXML, however, to aid the checking of these numbers there are a few predefined functions, arithmetic and function definition to achieve a higher level of complexity of validators.

7.1 Useful resources

What tools and languages exist and how can they be used to create CheckXML?

In Section 3.8 two main tools were discussed: Schematron and Relax NG. These tools validate XML. Schematron is closest to what CheckXML is, since elements can depend on each other, but they both lack computational dependency.

JastAdd was used to generate the compiler, with assistance from two other tools, JFlex (scanner generator) and Beaver (parser generator).

There are two mainly used tools for validating XML structure: XSD, DTD. Since DTD is a bit out dated XSD is used, but Relax NG would probably be preferred since it is easier to learn and use, allowing new users to adopt the language quicker.

It is possible to extract content from XML documents using DOM or SAX. These methods allows for a way to parse the XML document, which might be useful. However, in this case it is not needed since the goal is simply to extract the content. XPath does this well and thus it is used.
7.1.1 Lack of tools

There are no tool that does what CheckXML does. A mix of validation and computational interaction with XML. A reason for this might be that it is easy for a programmer to make a program do what CheckXML does in Java. So if a certain situation has required a tool like CheckXML it would have been a specific solution, by an experienced programmer. Instead CheckXML has focused on taking a step back and make easy to read and write validators with all the unnecessary surrounding code hidden from the user.

7.2 Implementation

RQ2: How to design and implementation a validation language to solve the test case?

Using aspect oriented programming with JastAdd a language was implemented, called CheckXML, to solve the test case. The implementation is based on scanning and parsing a program, then generating Java code that performs the validation according to the rules in a program. This generated code is a validator that takes XML as input and validates it against the rules from the program.

7.3 Testing

RQ3: How to make CheckXML’s syntax become easily readable?

The "think aloud" method worked well in this case as a method to improve the readability of CheckXML. The test subjects were able to work themselves through the test case and give valuable feedback. Some of the test subjects were a bit shy and had to be reminded that they were to speak what they were thinking.

The feedback was used to improve CheckXML. The syntax for a function was changed to DEFINITION since most test subjects used this word when trying to understand what they were reading. The syntax for declaring a variable DOUBLE was removed since it had no real purpose and was confusing for the reader.

Since the test subjects pretty much created an iteration in their head it seemed natural to implement iteration in CheckXML. This has other benefits as well, such as scaling and size reduction.

The improvements have made CheckXML easier to learn and more efficient.

When creating a readable language it is useful use few notations. The more new notations a user has to understand the harder a program is to interpret.

7.4 Future work

This section discusses work that could be done to further improve CheckXML.
7.4.1 Additional types

In the original version of CheckXML there were two types, Double and NodeSet. But when the test case only required numbers, NodeSet was dismissed and the notation for numbers were removed since every variable is a number. But it would be useful to have a type for nodes and strings. If there were a node type it could be used to send sets of nodes to functions and make computations, instead of sending individual values. This was partially solved with iterations since the FORALL iterates over a set of nodes.

This would allow for further use of XPath. XPath is powerful in picking sets of nodes from XML. This potential is lost without a Node type.

7.4.2 XSD to validate XPath

Since the XSD is available it would be useful to validate the XPath queries used in CheckXML against the XSD. In the current version the XPath is assumed to be correct, if it is not an exception will be thrown. A way to implement this would be to parse the XPath query and match it against the XSD.

7.4.3 XSD vs Relax NG

XSD is the easiest choice when it comes to validating the XML structure. But since Relax NG is simpler and easier to learn it might be better off since CheckXML is suppose to be available for many different systems. The simplicity would aid when deploying a new validator. So it would be worth investigating with user tests, similar to the one carried out in this thesis, if Relax NG is better than XSD.

7.4.4 Further testing

After the interviews were completed CheckXML was changed according to the conclusions of the results from the interviews. It would be useful to make further testing to see if the changes had positive effect and to make further improvements.
7. CONCLUSION


Appendices
Appendix A
Chunks of code

A.1 XML input for test case

Listing A.1: The XML document from test case

```xml
<Message>
  <SourceLocation>
    <Coordinates>
      <!-- Angelholm -->
      <Lat>56.2523745</Lat>
      <Long>12.866842</Long>
      <Alt>13</Alt>
    </Coordinates>
    <Time>1425029234</Time>
  </SourceLocation>
  <Destination>
    <Coordinates>
      <!-- Lund -->
      <Lat>55.7068014</Lat>
      <Long>13.197996</Long>
      <Alt>13</Alt>
    </Coordinates>
    <Time>1425045600</Time>
  </Destination>
  <WayPoints>
    <WayPoint>
      <Coordinates>
        <!-- Helsingborg -->
        <Lat>56.0352628</Lat>
      </Coordinates>
    </WayPoint>
    <WayPoint>
      <!-- Lund -->
      <Lat>55.7068014</Lat>
      <Long>13.197996</Long>
      <Alt>13</Alt>
    </WayPoint>
  </WayPoints>
</Message>
```
A.2 Full parser for CheckXML

Listing A.2: Full parser of CheckXML

%terminals ASSIGN, LCURLY, RCURLY, LPAREN, RPAREN, CHECK;
%terminals RETURN, FUNCTION, COMMA;
%terminals PATH, NUMBER, ID, SCHEMALINE, COMPARE;
%terminals ADD, SUB, MUL, DIV, MOD;

%left MUL, DIV, MOD;
%left SUB;
%right ADD;

%goal program;

program = schemaline_list_eps.c func_list_eps.fle
              stmt_list_eps.ele
            {: return new Program(c, fle, ele); :};
schemaline_list_eps =
/* epsilon */ { return new List(); }

schemaline_list = schemaline.a { return new List().add(a); }
| schemaline_list.al schemaline.a
{ return al.add(a); }
;
schemaline = SCHEMALINE.c { return new SchemaLine(c); }
;
func_list_eps = /* epsilon */ { return new List(); }
| func_list
;
func_list = func.a { return new List().add(a); }
| func_list.al func.a
{ return al.add(a); }
;
func = FUNCTION funcdecl.idd LPAREN arg_list_eps.ale RPAREN LCURLY stmt_list_eps.sle RCURLY
{ return new Func(idd, ale, sle); }
;
arg_list_eps =
/* epsilon */ { return new List(); }
| arg_list
;
arg_list = decl.a { return new List().add(a); }
| arg_list.al COMMA decl.a { return al.add(a); }
;
stmt_list_eps =
/* epsilon */ { return new List(); }
| stmt_list
;
stmt_list = stmt.a { return new List().add(a); }
| stmt_list.al stmt.a { return al.add(a); }
;
stmt = decl | check | return | forall;
decl = iddecl.i init.init { return new Decl(i,init); }
;
check = CHECK data.a compare.c data.b
{ return new Check(a,c,b); }
;
return = RETURN data.d { return new Return(d); }
;
forall = FORALL iddecl.id IN path.p except.opte LCURLY
stmt_list_eps.esl RCURLY
{ return new ForAll(id, p, new Opt(opte), esl); }
;
except = /* Epsion */ { return new Except(); }
| EXCEPTFIRST.ef { return new Except(ef); }
A.3 Test case solution before testing

Listing A.3: Full program example of CheckXML used for testing

FUNCTION DOUBLE haversineFunction(DOUBLE diff) {
    DOUBLE haversine = (1-cos(diff)) / 2
    RETURN haversine
}
FUNCTION DOUBLE
distanceBetweenCoords(DOUBLE long1, DOUBLE lat1,
DOUBLE long2, DOUBLE lat2) {
    DOUBLE earthRadius = 6371
    DOUBLE squareRoot = sqrt( haversineFunction(lat2-lat1) +
        cos(lat1)*cos(lat2)*
        haversineFunction(long2-long1) )
    DOUBLE result = 2 * earthRadius * arcsin(squareRoot)
    RETURN result
}

DOUBLE sourceLat = #/Message/SourceLocation/
    Coordinates/Lat#
DOUBLE sourceLong = #/Message/SourceLocation/
    Coordinates/Long#
DOUBLE way1Lat = #/Message/WayPoints/
    WayPoint[1]/Coordinates/Lat#
DOUBLE way1Long = #/Message/WayPoints/
    WayPoint[1]/Coordinates/Long#
DOUBLE sourceTime = #/Message/SourceLocation/Time#
DOUBLE way1Time = #/Message/WayPoints/WayPoint[1]/Time#

DOUBLE distance = distanceBetweenCoords(sourceLong, sourceLat,
    way1Long, way1Lat)
DOUBLE time = way1Time - sourceTime
DOUBLE timeInHours = time / 60 / 60
DOUBLE speed = distance / timeInHours
CHECK speed <= 120

DOUBLE way2Lat = #/Message/WayPoints/WayPoint[2]/Coordinates/Lat#
DOUBLE way2Long = #/Message/WayPoints/WayPoint[2]/Coordinates/Long#
DOUBLE way2Time = #/Message/WayPoints/WayPoint[2]/Time#
DOUBLE distance2 = distanceBetweenCoords(way1Long, way1Lat,
    way2Long, way2Lat)
DOUBLE time2 = way2Time - way1Time
DOUBLE timeInHours2 = time2 / 60 / 60
DOUBLE speed2 = distance2 / timeInHours2
CHECK speed2 <= 120
A.4 Test case solution

Listing A.4: Full solution for the test case written in CheckXML after changes

```xml
<xs:schema xmlns:xz="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Message">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="WayPoints">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="Coordinates" />
              <xs:element name="Time" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
type="xs:integer" />

</xs:sequence>
</xs:complexType>
</xs:element>

</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="Coordinates">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Lat">
    <xs:simpleType>
     <xs:restriction base="xs:double">
      <xs:minInclusive value="-90"/>
      <xs:maxInclusive value="90"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="Long">
    <xs:simpleType>
     <xs:restriction base="xs:double">
      <xs:minInclusive value="-180"/>
      <xs:maxInclusive value="180"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="Alt">
    <xs:simpleType>
     <xs:restriction base="xs:double">
      <xs:minInclusive value="-10000"/>
      <xs:maxInclusive value="10000"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
DEFINITION haversineFunction(diff) {
    haversine = (1-cos(diff)) / 2
    RETURN haversine
}

DEFINITION distanceCoords(long1, lat1, long2, lat2) {
    squareRoot = sqrt( haversineFunction(lat2-lat1) +
                      cos(lat1)*cos(lat2)*
                      haversineFunction(long2-long1) )
    result = 2 * 6371 * arcsin(squareRoot)
    RETURN result
}

FORALL waypoint IN #/Message/WayPoints/WayPoint# EXCEPTFIRST
    prevLat = PREV waypoint#Coordinates/Lat#
    prevLong = PREV waypoint#Coordinates/Long#
    currLat = waypoint#Coordinates/Lat#
    currLong = waypoint#Coordinates/Long#
    prevTime = PREV waypoint#Time#
    currTime = waypoint#Time#
    distance = distanceCoords(prevLong, prevLat, currLong, currLat)
    time = currTime - prevTime
    timeInHours = time / 60 / 60
    speed = distance / timeInHours

    CHECK speed <= 120
}