

**AUTOMATIC GENERATION OF MULTI-LANGUAGE
OBJECT DOMAIN MODELS THROUGH A SHAPE
EXPRESSIONS SUBSET**

GUILLERMO FACUNDO COLUNGA

Supervisors: Jose Emilio Labra Gayo and Daniel Fernández Álvarez.

A DISSERTATION SUBMITTED FOR THE DEGREE OF SOFTWARE
ENGINEER

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF OVIEDO

2020

Acknowledgments

This work has been partially funded by the European Hercules project through the WESO research group at the University of Oviedo.

In addition, for the development of the research we have worked with different research groups and researchers such as Dublin Core or Eric Prud'hommeaux to whom I would like to thank the warm welcome.

Finally, I would like to thank my tutors Jose Emilio Labra Gayo and Daniel Fernández Álvarez for all the work carried out within the framework of this project.

Abstract

Huge volumes of data are generated every day at high speeds. In addition, these data tend to come from very varied sources and therefore have heterogeneous structures. RDF was proposed as a graph-based data model. It relies on the global nature of URIs to solve the integration of produced data by different sources. However RDF by itself does not allow to validate that a graph complies with a contract or structure. That is why the technology called Shape Expressions was born. Through Shape Expressions an RDF graph can be validated against a certain schema. So with a set of Shape Expressions you can describe a domain model of a RDF data set. This technology is supported by an active community that has developed tools such as development environments, RDF validators, inference systems, etc ...

In this work, on one hand, a static analysis system is proposed. From which up to 90% of current Shape Expressions users can benefit. It is based on lexical, syntactic and semantic analysis that improves the detection and reporting system for errors and warnings. On the other hand, as Shape Expressions has as purpose to model a domain and this is the same purpose as what the object-oriented programming languages have. We provide a System to automatically translate shape expressions into object models of different object oriented languages. With that system, we have been able to transform 50% of all existing shape expressions in GitHub to Java classes. The translated shapes are those ones that are syntactically valid and stick to some language restrictions.

Keywords — *RDF, Linked Data, RDF Validation, Shape Expressions, Lexical-Syntactic and Semantic Validator, Object Oriented Programming Languages, Compiler, Translator.*

Contents

List of Figures	viii
List of Tables	xi
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	4
1.3 Structure of the Document	4
2 Theoretical Background	6
2.1 RDF	6
2.2 Validating RDF	7
2.2.1 Shape Expressions	8
2.2.2 Other Technologies	11
2.3 Programming Languages	11
2.4 Compilers	12
2.4.1 Internal Structure	12
2.4.2 Conventional Compilers	13
2.4.3 Modern Compilers	13
3 Related Work	14
3.1 Simplifications of ShEx	14
3.1.1 The S language	14
3.1.2 ShExJ Micro Spec	14
3.2 ShEx Ecosystem Tools	15
3.2.1 Validators	15
3.2.2 IDEs	15
3.2.3 Others	16
I Enhancing Error and Warning Detection and Reporting on ShEx	18
4 Analysis of Existing Syntactic and Semantic Analysers	19
4.1 Methodology	19
4.2 Syntactic Analysers	20
4.3 Semantic Analysers	21
4.4 Possible Enhancements	22

5	Proposed Syntactic and Semantic Analyser	23
5.1	Error Handler	23
5.2	Lexical Analyser	24
5.3	Syntactic Analyser	26
5.4	Semantic Analyser	28
5.5	Full System Diagram	29
6	Proposed Implementation	30
6.1	Structure	30
6.1.1	Parser	30
6.1.2	Syntactic Analyser	31
6.1.3	Semantic Analyser	32
6.2	Implementation	32
6.2.1	Parser	33
6.2.2	Syntactic Analyser	33
6.2.3	Semantic Analyser	34
6.3	Tests	35
6.4	Syntactic and Semantic Error and Warnings Detected	35
6.4.1	Not trailing semicolon at last triple constraint	36
6.4.2	Prefix not defined	36
6.4.3	Shape not defined	36
6.4.4	Prefix overridden	37
6.4.5	Shape overridden	37
6.4.6	Unused prefix definition	38
6.4.7	Base set but not used	38
II	Translating ShEx Schemas to Object Domain Models	39
7	Object Domain Model Translation Problem	40
7.1	Shape Expressions Expressivity	41
7.2	Plain Objects Expressivity	42
7.2.1	Plain Objects Structure	42
7.2.2	Plain Objects Formalization	43
7.2.3	Plain Objects Language Expressivity Dependence	44
7.2.4	Plain Objects Expressivity Generalization	44
7.3	Shape Expressions and Plain Objects Expressivity Comparison	45
8	Proposed Translator	48
8.1	Translator Formalization	48
8.2	Translator modelling	49

9	Proposed Implementation	52
9.1	Translator Back-end	52
9.2	Tests	54
9.3	Generated Objects	54
9.3.1	Real (Hercules ASIO European Project)	54
9.3.2	Synthetic (Generated)	55
III	Project Synthesis	57
10	Evaluation of Results	58
10.1	Methodology	58
10.2	Datasets	58
10.3	Results	58
11	Planning and Budget	62
11.1	Planning	62
11.1.1	Presentation of the Proposal	62
11.1.2	Presentation of the Dissertation	62
11.1.3	Defence of the Work	63
11.2	Budget	64
11.2.1	Proposal Preparation	64
11.2.2	Research	65
11.2.3	Development	65
11.2.4	Aggregated Costs	65
12	Conclusions	67
12.1	Future Work	67
12.1.1	Implementation of New Analyses	68
12.1.2	Automatic Error Fixing	68
12.1.3	Machine Learning to Recognize Patters	68
12.1.4	New Input Syntaxes for Translation to Aspect Oriented Languages	68
12.1.5	New Target Languages	68
IV	Annexes and References	69
Appendix A	ShEx Micro Language	70
A.1	Syntax Specification	70
A.2	Lexical Specification	70
Appendix B	ShEx-Lite Antlr Grammar	72
B.1	Syntax Specification	72

List of Figures

1.1	The 5 star steps of Linked Data	2
2.1	RDF N-Triples Example	6
2.2	RDF N-Triples Graph Example	7
2.3	RDF Example graph	7
2.4	RDF node and its shape	8
2.5	Shape Expression Example	8
2.6	Shapes, shape expression labels and triple expressions	9
2.7	Parts of a triple expression	9
2.8	Compiler stages	12
3.1	ShEx-Lite integration with Shexer	17
4.1	Examples of ShEx micro Compact Syntax code containing syntactic and semantic errors or warnings	20
5.1	Error handler use cases	23
5.2	Error handler functional requirements	24
5.3	Error handler non functional requirements	24
5.4	Error handler component and class diagrams	25
5.5	Lexical analyser use cases	25
5.6	Lexical analyser functional requirements	25
5.7	Lexical analyser non functional requirements	26
5.8	Lexical analyser component and class diagrams	26
5.9	Syntactic analyser use cases	26
5.10	Syntactic analyser functional requirements	27
5.11	Syntactic analyser non functional requirements	27
5.12	Syntactic analyser component and class diagrams	27
5.13	Semantic analyser use cases	28
5.14	Semantic analyser functional requirements	28
5.15	Semantic analyser non functional requirements	28
5.16	Semantic analyser component and class diagrams	29
5.17	Complete system class diagram	29
6.1	Syntactic and Semantic Analyser structure	30
6.2	Syntax Tree twenty first nodes produced by the parser	31

B.2 Lexical Specification	74
Appendix C GitHub CI Workflow	77
References	79

6.3	Abstract Syntax Tree produced after validation and transformations	32
6.4	Checker implementation for missing semicolons warning generation	33
6.5	Syntactic warning produced by the proposed Syntactic analyser	34
6.6	Common information stored at any AST node	34
6.7	Semantic error produced by an undefined prefix	36
6.8	Semantic error produced by an undefined shape	37
6.9	Semantic error produced by a prefix override	37
6.10	Semantic error produced by a shape override	38
6.11	Semantic warning produced by a prefix never used	38
6.12	Semantic warning produced by a base set but never used	38
7.1	Schema modeling a Person in ShExC syntax to the left. And the expected translated code in Java to the right.	40
7.2	ShEx Micro Abstract Grammar	41
7.3	Shape expression modeling the properties of a Person	42
7.4	Java, Python and Rust codings of Person object.	42
7.5	Java plain object decomposition.	43
7.6	Rust struct modeling a Person to the left. And the most similar approximation in Java to the right. In the Java approximation the Pet class is an interface that it is inherited by the Cat and Dog classes, that way we allow to store in the variable owningPet values of type Cat and Dog	44
7.7	Plain Objects Partial Generalization	45
7.8	Plain Objects Complete Generalization	45
7.9	Mapping function from ShEx to Plain Object	46
8.1	Different target types generated by specific translators	49
8.2	Translator use cases	49
8.3	Translator high level flowchart	50
8.4	Translator functional requirements	50
8.5	Translator non functional requirements	50
8.6	Translator component and class diagrams	51
9.1	Translator generic structure	52
9.2	Class diagram of the code generation visitors	53
9.3	CLI menu of ShEx-Lite CLI tool	54
9.4	Schema modelling a University in shex1 syntax to the left. And the ShEx-Lite generated code in Java to the right.	55
9.5	Schema modelling a Researcher in shex1 syntax to the left. And the ShEx-Lite generated code in Java to the right.	55
9.6	Synthetic schema in shex1 syntax to the left. And the ShEx-Lite generated code in Java to the right.	56

10.1	Bar chart for Table 10.1	59
10.2	Semantic error produced by ShEx-Lite for an undefined prefix	60
10.3	Semantic error produced by RDFShape and YASHE for an undefined prefix	60
10.4	Semantic error produced by RDFShape for an undefined prefix	61
11.1	Project timeline	62
11.2	Tasks planning of the project	63
11.3	Proposal preparation costs	64
11.4	Research costs	65
11.5	Development costs	65
11.6	Aggregated costs	66

List of Tables

4.1	Detection of the different syntactic errors by the current existing ShEx tools that syntactically analyse the shape expressions.	21
4.2	Detection of the different semantic errors by the current existing ShEx tools that semantically analyse the shape expressions.	21
10.1	Results produced for synthetic tests 1-13	59
10.2	Comparison of information provided in error and warning messages	60
10.3	Values obtained after compiling all the elements of our real case dataset with the ShEx-Lite system	61
11.1	Statistics of the main project tasks	64

CHAPTER 1

Introduction

This chapter covers the motivation, contributions and structure of the document. The main objective of this chapter, therefore, is that after reading it, the reader builds an idea about the motivations that have promoted this project, what is being worked on and the contributions emanating from it.

1.1 Motivation

Each day, more and more devices generate data both automatically and manually, and also each day the development of applications in different domains that are backed by databases and expose these data to the web becomes easier. The amount and diversity of data produced clearly exceeds our capacity to consume it.

To describe the data that is so large and complex that traditional data processing applications can't handle the term Big Data [1, 2] has emerged. Big Data has been described by at least three words starting by V: volume, velocity, variety. Although volume and velocity are the most visible features, variety is a key concept which prevents data integration and generates lots of interoperability problems.

RDF (*Resource Description Framework*) was proposed as a graph-based data model [3] which became part of the Semantic Web [4] vision. Its reliance on the global nature of URIs¹ offered a solution to the data integration problem as RDF datasets produced by different means can seamlessly be integrated with other data.

Related to this, is the concept of Linked Data [5] that was proposed as a set of best practices to publish data on the Web. It was introduced by Tim Berners-Lee and was based on four main principles, as mentioned in [5]:

- Use URIs as names for things.

¹A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource. To guarantee uniformity, all URIs follow a predefined set of syntax rules, but also maintain extensibility through a separately defined hierarchical naming scheme. Ref.https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

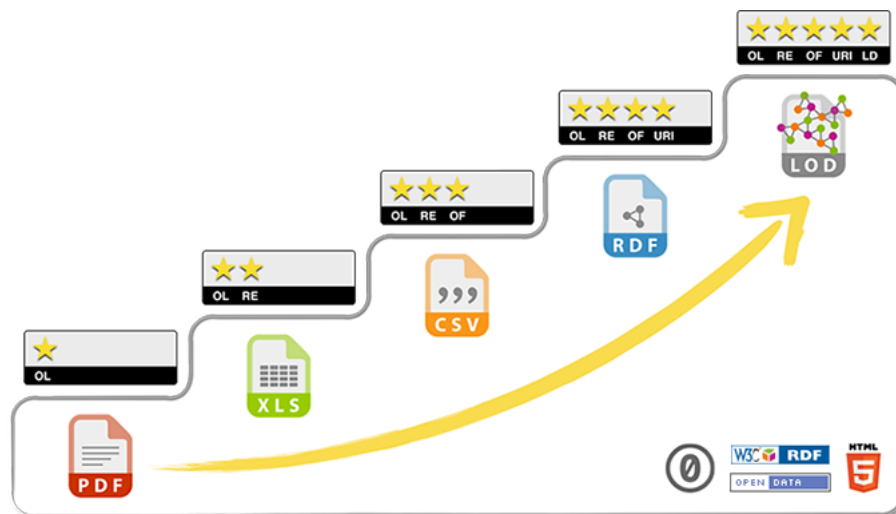


Figure 1.1: The 5 star steps of Linked Data.

- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs. so that they can discover more things.

Similar to this four principles is the 5 stars Linked Open Data Model, illustrated in [Figure 1.1](#). RDF is mentioned in the third principle as one of the standards that provides useful information. The goal of this principles is that data is not only ready for humans to navigate through but also for other agents, like computers, that may automatically process that data.

All the above motivations helped to make RDF the language for the Web of Data, as described in [6]. And the main features that it presents are: *Disambiguation*, *Integration*, *Extensibility*, *Flexibility* and *Open by Default*. With the features also some drawbacks are associated, the most important one and the one we will focus is the RDF **production/consumption dilemma**.

RDF production/consumption dilemma states that it is necessary to find ways that data producers can generate their data so it can be handled by potential consumers. For example, they may want to declare that some nodes have some properties with some specific values. Data consumers need to know that structure to develop applications to consume the data.

Although RDF is a very flexible schema-less language, enterprise and industrial applications may require an extra level of validation before processing for several reasons like security, performance, etc.

To solve that dilemma and as an alternative to expecting the data to have some structure

without validation, Shape Expressions Language (*ShEx*) was proposed as a human-readable and high-level open source language for RDF validation. Initially, ShEx was proposed as a human-readable syntax for OSLC Resource Shapes [7] but ShEx grew very fast to embrace more complex user requirements coming from clinical and library use cases.

Another technology, SPARQL Inferencing Notation (SPIN) [8], was used for RDF validation, principally in TopQuadrant's TopBraid Composer. This technology, influenced from OSLC Resource Shapes as well, evolved into both a private implementation and open source definition of the SHACL (*Shapes Constraint Language*), which was adopted by the W3C Data Shapes Working Group.

From a user point of view the possibilities of ShEx are very large, from the smallest case to just validate a node with one property to a scientific domain case where we need to validate the human genome². A language with such a number of possibilities requires from a strong syntactic and semantic validation and that leads us to our first goal.

Project Goal 1. *Determine how much the existing syntactic and semantic validation systems for shape expressions can be enhanced. And if existing systems can be enhanced propose a prototype that implement those enhancements.*

Secondly and very related to programming languages, if we take the Popularity of Programming Language (*PYPL*) Index³ from June 2020 we can see that more than half of the share is occupied by languages that support the object oriented paradigm. And therefore this paradigm becomes the most used one. The aim of this paradigm is to model real world domains, according to [9]. That, in fact, is the same goal that ShEx has, it allows to model real world domains with schemas, and validate existing data with them. Therefore our second goal relies on this and tries to automatically transform shape expressions into object domain models coded in any language that supports the object oriented paradigm:

Project Goal 2. *Determine till which point can we automatically translate existing shape expressions to object domain models. And propose a prototype capable of translating Shape Expressions to object domain models.*

If this were possible it would not only imply that you could automate the creation of application domain models but that you could link the domain model that an application uses with a domain model defined through Shape Expressions that describes the schema of a RDF data set.

To give answers to the questions posed in this section, we will limit our scope to the micro grammar of Shape Expressions, defined in ⁴. This version is a strict subset of the complete

²<https://github.com/geneontology/go-shapes>

³<http://pypl.github.io/PYPL.html>

⁴https://dcmi.github.io/dcap/shex_lite/micro-spec.html

ShEx grammar and therefore any derived method or technology we can draw from it can automatically be applied to the full grammar.

1.2 Contributions

These are the major contributions of this dissertation:

1. A parser for the ShEx micro Compact Syntax. There are already existing parsers for ShEx and they work for ShEx micro Compact Syntax as it is a subset of ShEx, but they accept more structures than the ones defined by ShEx micro Compact Syntax. We propose a parser that is only focused on ShEx micro Compact Syntax and therefore error and warning messages can be enhanced.
2. Error and warning analyser for schemas. Existing approaches do not semantically validate the schemas, they only perform error detection by means of complex grammars and parsers. Our proposed system does semantically validate the schemas by means of a custom analyser that performs both syntactic and semantic analysis so it produces human-friendly errors and warnings that users can use to fix their schemas.
3. Automatic translation of schemas into object domain models in `Java` and `Python`. The proposed system integrates an open back-end with built-in code translation from the validated schemas to domain models in Object Oriented Programming Languages (*OOPL*) [10].
4. Evaluation of errors and warnings generated of our proposed solution against existing tools. This comparison empirically shows the benefits and drawbacks of our proposed system.

1.3 Structure of the Document

The dissertation layout is as follows:

Chapter 2 Indicates the state of the art of the existing RDF validation technologies, tools for processing Shape Expressions and other related projects.

Chapter 3 Gives a basic theoretical background that it is needed to fully understand the concepts explained in the following chapters.

Chapter 4 Analyses current syntactic and semantic analysis systems.

Chapter 5 He proposes a system by means of software engineering techniques that tries to solve the problem posed in the previous chapter.

Chapter 6 It proposes an implementation that meets the expectations of the previous chapter. This implementation is the one that will be used to carry out the evaluation

of results.

Chapter 7 Define the ShEx translation problem to object-oriented languages and compare the expressivities of both systems.

Chapter 8 It focuses on proposing a solution to the problem raised in the previous chapter. First through formalizations and then employing software engineering methodologies.

Chapter 9 It offers a proposed implementation to meet the expectations of the previous chapters. This implementation will be used to evaluate the results.

Chapter 10 It defines a methodology and the data on which the methodology will be tested. Then evaluate the results obtained.

Chapter 11 It includes the description of the project planning as well as its cost budget.

Chapter 12 It summarizes the results achieved after completing the work and includes the proposals for future work.

CHAPTER 2

Theoretical Background

For a proper understanding of this documentation and the ideas explained on it, it is needed to know some theoretical concepts that are the fundamentals of Linked Data, RDF, RDF Validation, programming languages and compilers. For those readers that want a more detailed view of the concepts presented here is offered in [6, 11, 12].

2.1 RDF

Resource Description Framework (RDF) is a standard model for data interchange on the web. It started in 1998 and the first version of the specification was published in 2004 by the W3C according to [13]. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed. Another important feature is that RDF supports XML, N-Triples and Turtle syntax. The most common representation though is the N-Triples syntax. It is composed of a set of triples. Each triple is composed of three elements, the subject the predicate and the object. The [Figure 2.1](#) shows an example of how a triplet can be written in RDF N-Triples Syntax.

RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link, this is usually referred to as a “triple” or “triplet”. As we can see from [Figure 2.1](#) each triple is composed of three elements, the subject, the predicate and the object. Each one of them can contain any URI value. Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications. [Figure 2.3](#) shows an example of how different triples can be used to compose a graph, this graph represents the same as the [Figure 2.2](#)

```
1 <http://example/subject1> <http://example/predicate1> <http://example/object1> .
```

Figure 2.1: RDF N-Triples Example. From this example we can see that each triplet is composed of three elements, the subject the predicate and the object.

```
1 <http://example/bob> <http://example/knows> <http://example/alice> .  
2 <http://example/alice> <http://example/knows> <http://example/peter> .
```

Figure 2.2: RDF N-Triples Graph Example. This example shows the n-triples that generate the graph from [Figure 2.3](#).

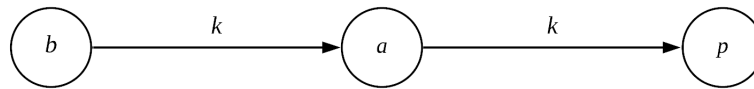


Figure 2.3: RDF graph formed by triplets from [Figure 2.2](#), where *b* corresponds to `<http://example/bob>`, *a* corresponds to `<http://example/alice>`, *p* corresponds to `<http://example/peter>` and *k* corresponds to `<http://example/knows>`.

This linking structure forms a directed, labelled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is usually used for an easy understanding of the RDF model.

Also, related to this we strongly recommend the Tim Berners-Lee's writings on Web Design Issues [14] where he explain the issues of the linked data and why is RDF so important.

2.2 Validating RDF

RDF therefore allows to represent and store data, and with this ability emerges the need to validate that the schema of the graph is correct. In order to perform the validation of RDF data there have been previous attempts, described in [Section 2.1](#), this dissertation will focus on Shape Expressions. But, in order to validate RDF data, every technology needs to face the following RDF concepts:

- The form of a node (the mechanisms for doing this will be called “node constraints”).
- The number of possible arcs incoming/outgoing from a node.
- The possible values associated with those arcs.

[Figure 2.3](#) illustrates those RDF concepts by means of the Shape Expression that validates users. There we can see that the shape of the RDF node that represents Users represents the form of a node, the number of possible arcs and the possible value associated with those arcs.

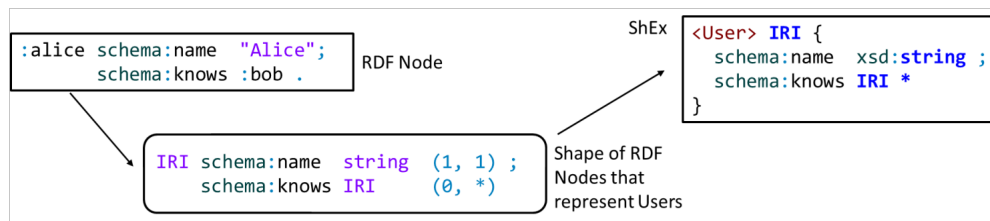


Figure 2.4: RDF node and its shape.

```

1 PREFIX :          <http://example.org/>
2 PREFIX schema:    <http://schema.org/>
3 PREFIX xsd:       <http://www.w3.org/2001/XMLSchema#>
4
5 :User {
6   schema:name          xsd:string   ;
7   schema:birthDate    xsd:date?    ;
8   schema:gender       [ schema:Male schema:Female ] OR xsd:string ;
9   schema:knows        IRI @:User*
10 }

```

Figure 2.5: Shape Expression Example. This example describes a shape expression that models a user as a node that has one name of type string, an optional birth date of type date, one gender of type Male, Female or free string and a set between 0 and infinite of other users represented by the knows property.

2.2.1 Shape Expressions

As defined in [6], Shape Expressions (ShEx) is a schema language for describing RDF graphs structures. ShEx was originally developed in late 2013 to provide a human-readable syntax for OSLC Resource Shapes. It added disjunctions, so it was more expressive than Resource Shapes. Tokens in the language were adopted from Turtle and SPARQL with tokens for grouping, repetition and wildcards from regular expression and RelaxNG Compact Syntax [15]. The language was described in a paper [11] and codified in a June 2014 W3C member submission, which included a primer and a semantics specification. This was later deemed “ShEx 1.0”.

As of publication, the ShEx Community Group was starting to work on ShEx 2.1 to add features like value comparison and unique keys. See the ShEx Homepage <http://shex.io/> for the state of the art in ShEx. A collection of ShEx schemas has also been started at <https://github.com/shexSpec/schemas>.

ShEx Compact Syntax: ShExC

The ShEx compact syntax (ShExC) was designed to be read and edited by humans. It follows some conventions which are similar to Turtle or SPARQL.

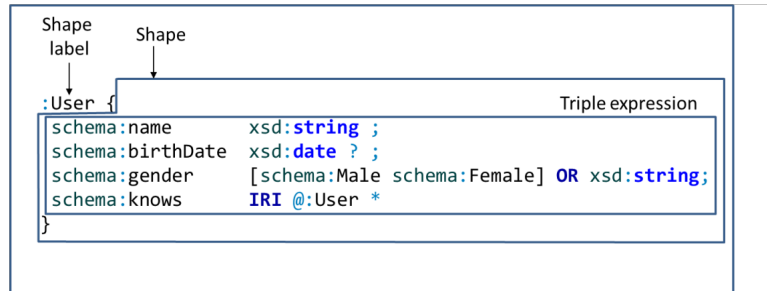


Figure 2.6: Shapes, shape expression labels and triple expressions.

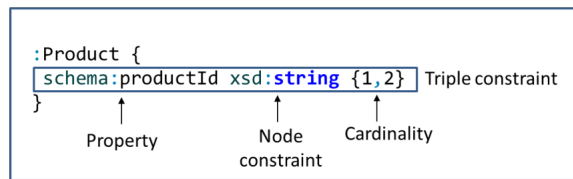


Figure 2.7: Parts of a triple expression.

- PREFIX and BASE declarations follow the same convention as in Turtle. In the rest of this chapter we will omit prefix declarations for brevity.
- Comments start with a # and continue until the end of line.
- The keyword a identifies the `rdf:type` property.
- Relative and absolute IRIs are enclosed by `< >` and prefixed names (a shorter way to write out IRIs) are written with prefix followed by a colon.
- Blank nodes are identified using `_:label` notation.
- Literals can be enclosed by the same quotation conventions (`'`, `"`, `'''`, `"""`) as in Turtle.
- Keywords (apart from a) are not case sensitive. Which means that `MinInclusive` is the same as `MININCLUSIVE`.

A ShExC document declares a ShEx schema. A ShEx schema is a set of labelled shape expressions which are composed of node constraints and shapes. These constrain the permissible values or graph structure around a node in an RDF graph. When we are considering a specific node, we call that node the focus node.

Figure 2.6 shows the first level of a shape expression, we have a label and the shape itself that is what we assign to the `:User` label. Then, the shape is composed by triple expressions. The triple expression structure is explained in Figure 2.7, and as its name indicates it is composed of three elements, the property, the node constraint and the cardinality.

Shape Expressions Compact Syntax is much bigger and contains other multiple features that give ShEx its power, and all of them can be explored in [6] but they are not needed to understand this dissertation.

Use of ShEx

Strictly speaking, a ShEx schema defines a set of graphs. This can be used for many purposes, including communicating data structures associated with some process or interface, generating or validating data, or driving user interface generation and navigation. At the core of all of these use cases we have the notion of conformance with schema. Even when someone is using ShEx to create shapes, the goal is to accept and present data which is valid with respect to a schema. ShEx has several serialization formats:

- a concise, human-readable compact syntax (ShExC);
- a JSON-LD syntax (ShExJ) which serves as an abstract syntax; and
- an RDF representation (ShExR) derived from the JSON-LD syntax.

These are all isomorphic and most implementations can map from one to another. Tools that derive schemas by inspection or translate them from other schema languages typically generate ShExJ. Interactions with users, e.g., in specifications are almost always in the compact syntax ShExC. As a practical example, in HL7 FHIR¹, ShExJ schemas are automatically generated from other formats, and presented to the end user using compact syntax.

ShExR allows to use RDF tools to manage schemas, e.g., running a SPARQL query to find out whether an organization is using `dc:creator` with a `string`, a `foaf:Person`, or even whether an organization is consistent about it.

ShEx Implementations

At the time of this writing, we are aware of the following implementations of ShEx.

- shex.js for Javascript/N3.js (Eric Prud'hommeaux) <https://github.com/shexSpec/shex.js/>;
- Shaclex for Scala/Jena (Jose Emilio Labra Gayo) <https://github.com/labra/shaclex/>;
- shex.rb for Ruby/RDF.rb (Gregg Kellogg) <https://github.com/ruby-rdf/shex>;
- Java ShEx for Java/Jena (Iovka Boneva/University of Lille) <https://gforge.inria.fr/projects/shex-impl/>; and
- ShExkell for Haskell (Sergio Iván Franco and Weso Research Group) <https://github.com/weso/shexkell>.

¹<https://www.hl7.org/fhir/>

There are also several online demos and tools that can be used to experiment with ShEx.

- shex.js (<http://rawgit.com/shexSpec/shex.js/master/doc/shex-simple.html>);
- Shaclex (<http://shaclex.herokuapp.com>); and
- ShExValidata (for ShEx 1.0) (<https://www.w3.org/2015/03/ShExValidata/>).

2.2.2 Other Technologies

As other validation technologies it is very interesting to know how other tools approach the same issue.

SHACL

From [6], Shapes Constraint Language (SHACL) has been developed by the W3C RDF Data Shapes Working Group, which was chartered in 2014 with the goal to “produce a language for defining structural constraints on RDF graphs [7].”

The main difference that made us choose ShEx over SHACL is that ShEx emphasized human readability, with a compact grammar that follows traditional language design principles and a compact syntax evolved from Turtle.

JSON Schema

JSON Schema born as a way to validate JSON-LD, and as turtle and RDF can be serialized as JSON-LD, it is usual to think that JSON Schema can validate RDF data, but this is not fully correct. And the reason is that the serialization of RDF data in to JSON-LD is not deterministic, that means that a single schema might have multiple serializations, which interferes with the validation as you cannot define a relative schema.

2.3 Programming Languages

According to [12], “a programming language is a formal language comprising a set of instructions that produce various kinds of output.” When we talk about programming languages we need to know that they are split into two categories, General Purpose Languages (GPL) and Domain Specific Languages (DSL). The main difference overtime is that, as stated in [16], a domain-specific language (DSL) is a computer language specialized to a particular application domain in contrast to a general-purpose language (GPL), which is broadly applicable across domains.

In the specific case of ShEx-Lite, we will be talking about a Domain Specific Language, and, more specifically, a declarative one, we would classified it as a Declarative one, that means that it is not Touring Complete [17].

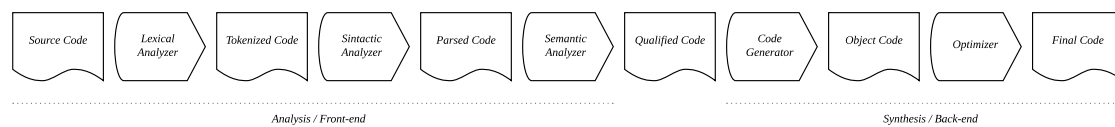


Figure 2.8: Compiler stages.

2.4 Compilers

A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). Is during this translation process where the compiler validates the syntax and the semantics of the program, if any error is detected in the process the compiler raises an exception (understand as a compiler event that avoids the compiler to continue its execution).

2.4.1 Internal Structure

In order to decompose the internal structure of a compiler, they have been split into the most common task they do (Figure 2.8). This does not mean that there are compilers that might have more or less stages. But at the end, everything can be grouped into any of the stages that we will explain:

Lexical Analyser

The lexical analyser task is to get the input and split it in to tokens [18], which are build from lexemes. If the compiler cannot find a valid token for some lexemes in the source code, it will generate an error, as the input cannot be recognized.

Syntactic Analyser

The syntactic analyser takes the tokens generated during the lexical analysis and try's to group the tokens so the conform to the language grammar rules. During this stage, if there is any error while trying to group the tokens, then the compiler will rise an error as the input cannot be parsed.

Semantic Analyser

The semantic analyser has two main tasks, usually. First, it validates that the source code semantics are correct. For example in java `4 + "aaa"` would not make sense as the semantic rules specify that and integer cannot be added to an string. And the second task is to transform the Abstract Syntax Tree in to a type-checked and annotated AST. Usually that means relate the invocations and variables to its definition, very useful for type-checking.

Code Generator

The task of the code generator as its name indicates is to generate the target code. It can be byte code, machine code, or even another high-language code.

Code Optimizer

The code optimizer is the last step before the final target code is generated, it rewrites the code that the code generator produced without changing the semantics of the program, its aim is just to make code faster. At [19] you can see an example of some optimizations that can be done at compile time to make your code faster.

2.4.2 Conventional Compilers

Conventional compilers are a big monolith where each stage 2.8 is executed automatically after the previous stage, if the compiler has eight steps you need to execute them all at once. This approach presents some drawbacks:

- A poor IDE [20] integration. IDE's need to perform incremental compilations in matter of nanoseconds so the user doesn't feel lag when typing the program. With conventional compilers as you need to go through all the compilation process at once they were very slow and companies like Microsoft need to develop different compilers, one for the IDE and another for the final compilation of the program itself. This led to several problems like that if a feature gets implemented in the final compilation compiler but not in the IDE one the IDE would not support the feature meanwhile the language would.
- Difficult to debug. As the conventional compilers were a black box the only way to test intermediate stages was by throwing an input and waiting the the feature you wanted to test was thrown for that input.

2.4.3 Modern Compilers

After the problems Microsoft had with the C# compiler they decide to rewrite the whole compiler and introduce a concept called "compiler as an API" with Roslyn [21]. This concept has been perfectly accepted and solved many problems, such as incremental compilation. In this concept each stage has an input and an output that can be accessed from outside the compiler, and stages can be executed independently on demand. For example if an IDE just want to execute the Lexer the Parser and the Semantic analysis it can. That translates in to speed for the user.

Also the second problem is solved as testing individual parts of the compiler is much more easy than the hole compiler at once.

CHAPTER 3

Related Work

Some work has already been done in the field of Shape Expressions and RDF validation technologies. In this chapter we will go over the main studies related to our project, exploring what they have achieved and some of their limitations.

3.1 Simplifications of ShEx

3.1.1 The S language

In 2019 at [22] was defined a language called **S** as a simple abstract language that captures both the essence of ShEx and SHACL. This is very relevant as this language is intended to be the input of a theoretical abstract machine that will be used for graph validation for both ShEx and SHACL. Also in the same paper the authors carefully describe the algorithm for the translation from ShEx to S and from SHACL to S.

Although the theoretical abstract machine has not been implemented yet the intention of the WESO Research Group, where this S language was defined, is to devote more efforts in to this project during the 2021.

Other definition of an abstract language based on uniform schemas can be found at [23]. This language is focused on schemas inference rather on validation, but needs to be taken into account as they also perform an abstraction of both ShEx and SHACL.

3.1.2 ShExJ Micro Spec

Recently the Dublin Core Team¹ is working into an specification that allows to define Shape Expressions in tabular formats. For this specification they propose a simplification of the Shape Expressions JSON syntax that allows to define an schema as a set of simple triple constraints. This specification is not official and has not been validated yet but it is very important for our work as we will also work with a similar simplification of a syntax of ShEx.

¹<https://dublincore.org/>

And to the best of our knowledge no other language based on a subset of Shape Expressions has been designed nor implemented yet.

3.2 ShEx Ecosystem Tools

We already know that ShEx and SHACL have been the two main technologies for RDF validation and some tools emerged around them, we think that some of them might benefit from ShEx-Lite. Here we introduce briefly those that had the biggest impact in the community.

3.2.1 Validators

Since the beginning of ShEx and SHACL as languages the RDF community started to build tools that take as input the schemas defined and validate graphs.

These tools can also benefit from the methods and technologies developed in this dissertation as they need to previously analyse the schemas before even try to validate RDF data with them.

The most important validators are:

Shaclex

According to the Shaclex² official website it is an Open Source Scala pure functional implementation of an RDF Validator that supports both Shape Expressions and SHACL. It was initially developed by Jose Emilio Labra Gayo and is being maintained by an active community on GitHub. It is used by different projects around the globe and its goal is to validate RDF graphs against schemas defined in Shape Expression or in SHACL.

This implementation of a ShEx validator is very important for us as ShEx-Lite is completely inspired by it and aims to transfer the syntactic and semantic validation enhancements to it.

ShEx.js

Another example of a ShEx validator implementation is `ShEx.js` which is JavaScript based and also open source on GitHub. This implementation is very important for the ShEx community as they defined the serialization of the AST in this implementation as the abstract syntax of ShEx.

3.2.2 IDEs

In order to facilitate the task of writing schemas some engineers decide to implement specific IDEs for the Shape Expressions Language.

²<https://github.com/weso/shaclex>

This tools can benefit from ShEx-Lite and there are currently collaborations in process, such as the integration of the unused resources detection in YASHE. At the time they work with Shaclex, which is structured as a conventional compiler, but with the API architecture of ShEx-Lite IDEs can access directly to the syntactic and semantic modules so features like advances colouring syntax or incremental compilation are available.

YASHE

YASHE³ (Yet Another ShEx Editor), is a Shape Expressions IDE which started as a fork of YASQE (which is based on SPARQL). This tool performs lexical and syntactic analysis of the content of the editor, thus offering the user a real-time syntactic error detector. It has features such as syntax highlighting, visual aid elements (tooltips) and autocomplete mechanisms. In addition, it offers a simple way of integrating into other projects.

Protégé

Protégé is a piece of software developed by the University of Stanford focused on ontology edition. During the last year they added support for Shape Expressions edition on their own software so they became another ShEx IDE.

VSCode

VSCode is a source code light-weight editor developed by Microsoft and supported by Linux, macOS and Windows. By default this editor does not support any programming language, the way it works is with packages that the community develops and extends the functionality. One of those packages adds support for Shape Expressions Compact syntax and transforms VSCode into a ShEx IDE.

This plugin does not add semantic validation and it is a clear target to benefit from ShEx-Lite features.

3.2.3 Others

Other researches focused their efforts in to inferring schemas to existing data sets and creating tools to that evolved from ShEx in order to transform existing data.

Shexer

Shexer⁴ is a python library aimed to perform automatic extraction of schemas in ShEx from an RDF input graph. This tool takes a graph, and from it, it infers the schemas that it might contain. Its work is fully described in [23, 24].

³<https://github.com/weso/YASHE>

⁴<https://github.com/DaniFdezAlvarez/shexer>

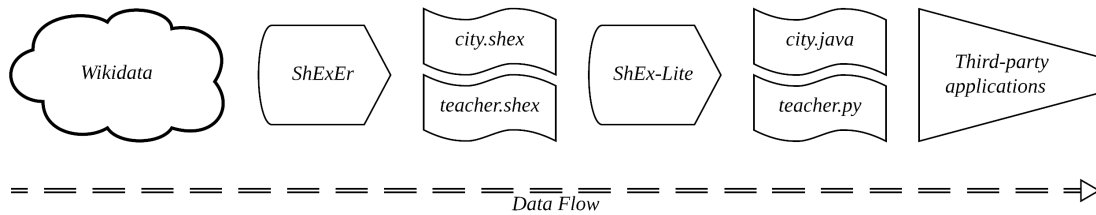


Figure 3.1: ShEx-Lite integration with ShExer for automatically generating java domain object models for the Wikidata schema less existing data. This shows the schema less data from Wikidata from which shape expressions are inferred by ShExer and later transformed to Java plain objects by means of ShEx-Lite so third-party applications can implement the domain model.

ShExML

ShExML⁵ is a language based on ShEx (not a simplification nor an abstraction of ShEx) that can map and merge heterogeneous data formats into a single RDF representation. The main idea behind this tool is written at [25].

An example of how these different tools can work together thanks to ShEx-Lite would be the following, illustrated at Figure 3.1. There we have Wikidata⁶ which is a free and open knowledge base that can be read and edited by both humans and machines. Wikidata, currently holds millions of registers that do not have any schema that validates them. And they need to make consumer applications that represent the data in to an object domain model. Without any tool this is just almost impossible, but with ShExer you can infer the schemas to ShEx-Lite syntax and with the ShEx-Lite compiler you can automatically create the object domain model in your favourite OOL.

⁵<https://github.com/herminiogg/ShExML>

⁶<https://www.wikidata.org>

Part I

Enhancing Error and Warning Detection and Reporting on ShEx

Analysis of Existing Syntactic and Semantic Analysers

In the Related Work ([Chapter 3](#)) some ShEx tools were explained. This section will detail more those tools that provide any kind error and warning detection and reporting. After, we will detail the points that we think can be enhanced.

Before start the analysis we must define a methodology in order to be able to make an even analysis for all existing tools.

4.1 Methodology

To evaluate existing systems from a neutral point of view we will use the ShEx specification as the basis. However, this specification does not cover all possible cases, in particular it leaves most semantic restrictions to the choice of the specific implementation.

Therefore, as regards this evaluation, when a semantic option not contemplated by the specification is proposed, the option that favours the security of the language will be chosen. For example. If the specification does not state anything about whether a variable can be redefined and we had to take an option, we will always choose not, so that the language is as safe as possible and does not lead to errors.

The unique syntactic restriction applied is:

- In the last triple constraint of a set expression the trailing semicolon it is optional but recommended.

The semantic restrictions that have been applied are listed below.

- Overwriting of prefixes is not allowed.
- Overwriting of the base is not allowed.
- Overwriting of the start shape is not allowed.
- Overwriting of shapes is not allowed.

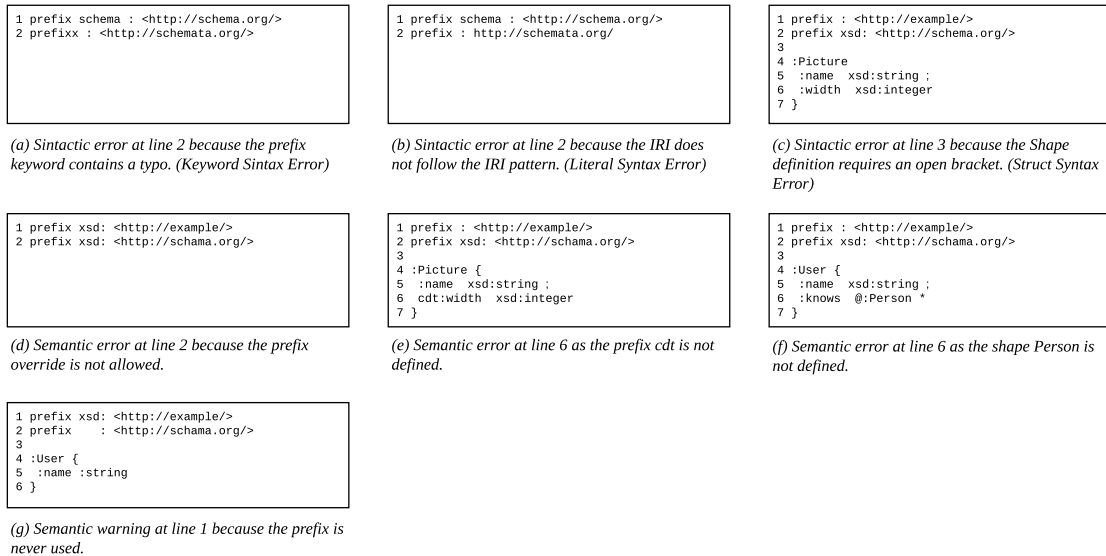


Figure 4.1: Examples of ShEx micro Compact Syntax code containing errors.

- All references must exist within the scope of the schema.

In addition, in this evaluation we will use different test cases for each system, specifically the test cases correspond to each element of the ShEx micro Compact grammar. Remember that the elements that this grammar has are: *definition of prefixes*, *definition of the base*, *definition of the start shape* and *definition of shapes*. You can also find references to prefixes, the base and other shapes. Therefore we will test all these elements in their syntactic and semantic aspects. [Figure 4.1](#) shows some examples of these errors.

4.2 Syntactic Analysers

According to [26] we consider a Syntactic Analyser a piece of software capable of parse, generate a parse tree and detect and retorting syntactic warnings and errors.

Therefore in this category we would include **Shaclex**, **ShEx.js**, **YASHE** and **VS Code Plugin**. [Table 4.1](#) shows a comparison between the analysed tools.

Some comments to be made about the results obtained are that although we get an error for syntactic errors, the quality of the error is more or less always the same. For example for the fragment `prefixx xsd: <http://example/>` where we introduced an error at the keyword `prefix` by adding an extra `x` the error obtained is: `This line is invalid. Expected: PNAME_NS.`

According to [27], this error message nor can be improve as it does not provide the user enough information to fix the schema.

Table 4.1: Detection of the different syntactic errors by the current existing ShEx tools that syntactically analyse the shape expressions.

Syntactic Errors								
Analysers	Prefix Definition	Base Definition	Start Shape	Shape Definition	Prefix Reference	Base Reference	Shape Reference	Recomends Semicolon Last Triple Constraint
Shaclex	Yes	Yes	Yes	Yes	Not completly	Yes	Yes	No
ShEx.js	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
YASHE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
VS Code Plugin	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

Table 4.2: Detection of the different semantic errors by the current existing ShEx tools that semantically analyse the shape expressions.

Semantic Errors								
Analysers	Prefix Override	Base Override	Start Shape Override	Shape Override	Non Existing	Non Existing	Non Existing	
					Prefix Reference	Base Reference	Shape Reference	
Shaclex	No	No	No	No	Yes	-	No	
ShEx.js	No	No	No	No	Yes	-	No	
YASHE	No	No	No	No	Yes	-	Yes ¹	

Then also it is important to remark that during this analysis we encounter other syntactic problems that where not detected by tools like Shaclex, an example is that properties like `schema:rdf@:name` (*which is not a valid IRI*) are accepted without errors.

4.3 Semantic Analysers

As Semantic Analysers we will only consider those tools that validate the semantics of the language, in this section we include the validation of references like prefixes and shapes. The tools that claim to support this validations are **Shaclex**, **ShEx.js**, and **YASHE**. Table 4.2 shows a comparison between the analysed tools.

From the obtained results we have to point that most of the tools opted for an open policy when talking about language semantics. From our point of view this have its advantages and its drawbacks. But this only affects to the override policy. All of the tools should implement the non existing references validation and most of them only focus on prefixes definition with the exception of YASHE which does the checking of the shape reference but the error message sometimes is not completely accurate.

It it also remarkable that none of the tools performs a deeper analysis so there is no detection of unused resources, therefore no warnings are generated by none of the existing tools.

4.4 Possible Enhancements

Previous sections show the current state of the existing tools, their capabilities and their lacks. With all that information we propose a list of enhancements that can be done to improve the error and warning detection. As seen in previous sections there's work that can be done to improve the existing ecosystem of tools. We have identified the following aspects that will benefit end users:

1. **Enhancement of error messages [27].** Existing error messages, originated both by syntactic or semantic errors do not offer information about the exact place that originates the error nor a processed description nor possible solutions.
2. **Creation of a new type of error messages with lower importance called warnings.** Currently systems do not analyse if declared resources are used and therefore there is no need to generate warnings. We propose to not only fully analyse the resources to detect non-used ones but also the creation of error messages with lower importance like warnings that can be used to offer more information to the end user.
3. **Detection of override definitions.** Most of the existing tools prefer not to detect when a definition is being overridden, we propose to detect those situations and treat definitions as fixed values.
4. **Detection of undefined references.** Some tools detect some broken references, we propose to encase this situation and take that behaviour to other elements like shape references.
5. **Detection of unused resources.** Related to the second point sometimes new users copy and paste old code which ends with lots of unused code, we propose a system that detects those situations and suggest to remove that unused code.
6. **Detection of multiple errors / warnings at once.** Most of the current analysers only provide information about the first error they find, this means that if we have a scheme with multiple errors or warnings, only the first one will be shown to us and we will not be able to see the next one until we solve the previous one.

CHAPTER 5

Proposed Syntactic and Semantic Analyser

After analysing the existing tools, we can see that different aspects of existing technology can be improved, such as those discussed in [Section 4.4](#). In this chapter we model a proposal by means of software engineering techniques.

Within these techniques, the process we are going to follow to model the proposal is first obtain the use cases through the possible improvements detected in the previous chapter. From these use cases we will extract the requirements. Once the requirements have been extracted, we will proceed to design the solution using diagrams.

We know that the system will be composed of at least a lexical analyser, a syntactic analyser, a semantic analyser and some type of message manager to handle errors and warnings.

5.1 Error Handler

Of the improvements that we observe in [Section 4.4](#), those that have to do with the error / warning management system are 1, 2 and 6. The following diagram considers these use cases in the error / warning management system.

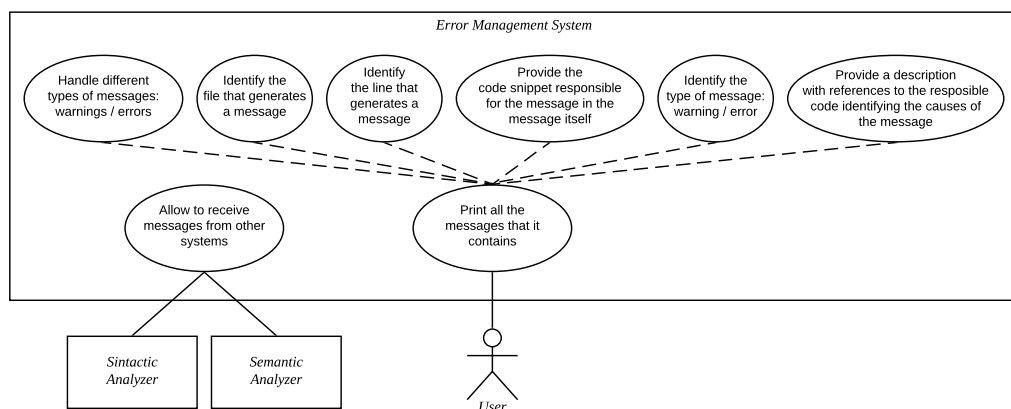


Figure 5.1: Error handler use cases.

Thus, from the use cases mentioned in the previous section we extract the following functional

requirements.

ID	DESCRIPTION
RF.1	The error management system must be able of handle multiple kind of messages.
RF.1.1	The error management system must support at least warnings and errors.
RF.2	The error management system must indicate the file where the message was generated
RF.3	The error management system must indicate the line where a message originates.
RF.4	The error management system must indicate the position on the line where a message originates.
RF.5	The error management system must indicate the code snippet in which a message originates.
RF.6	The error management system must indicate the type of the error such as described in Chapter 4.
RF.7	The error management system must indicate information to solve the error.
RF.7.1	In the case of syntactic errors, the reason for the error will be indicated.
RF.7.2	In the case of semantic errors, if the cause is conditioned by another element, a reference to this element must appear in the cause of the error.

Figure 5.2: Error handler functional requirements.

From the use cases we can also extract the following interface requirements.

ID	DESCRIPTION
RI.1	The error management system must be able of receiving messages from the syntactic analyzer.
RI.2	The error management system must be able of receiving messages from the semantic analyzer.

Figure 5.3: Error handler non functional requirements.

For the previous requirements we propose the following diagram for the error management system ([Figure 5.4](#)).

5.2 Lexical Analyser

The lexical analyser is necessary to subsequently carry out syntactic and semantic analysis. And therefore, although it is not contemplated as an improvement in the previous chapter, we do have to include it in our proposed system. The following diagram shows the expected use cases of a lexical analyser in our context.

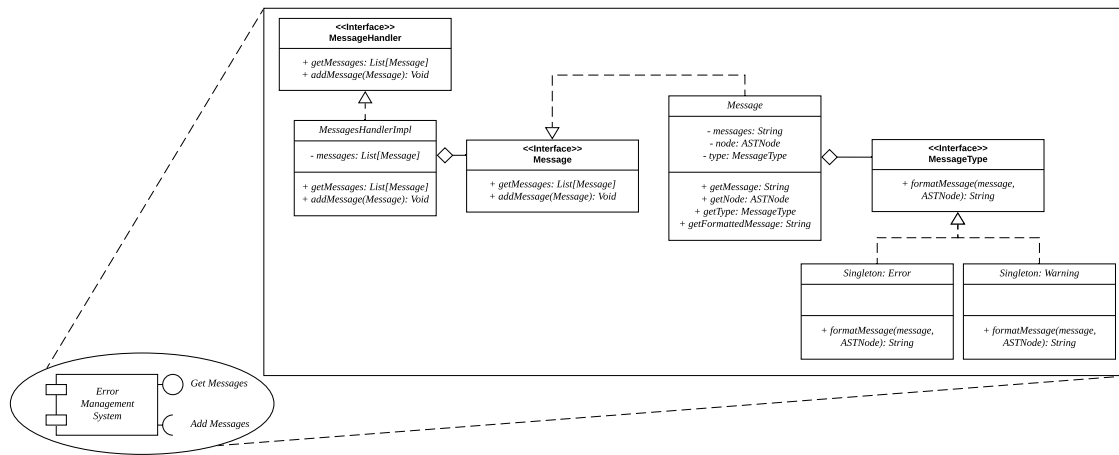


Figure 5.4: Error handler component and class diagrams.

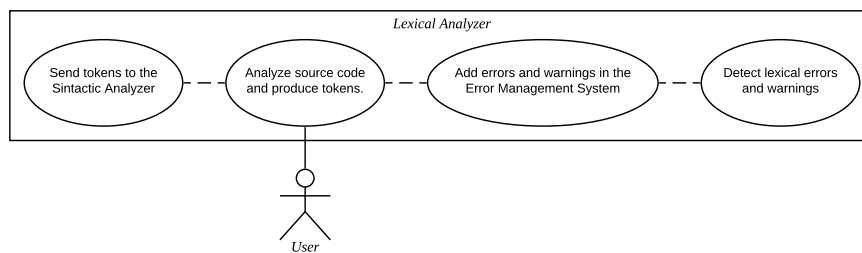


Figure 5.5: Lexical analyser use cases.

Thus, from the use cases mentioned in the previous section we extract the following functional requirements.

ID	DESCRIPTION
RF.8	The lexical analyzer must be able of analyze source code in the ShEx micro Compact syntax and produce tokens.
RF.9	The lexical analyzer must be able of detect lexical errors and warnings.
RF.9.1	A lexical error is produced when the input source code does not match the ShEx micro Compact lexical specification.

Figure 5.6: Lexical analyser functional requirements.

From the use cases we can also extract the following interface requirements.

For the previous requirements we propose the following model for the lexical analyser.

ID	DESCRIPTION
RI.3	The lexical analyzer must be able of send the produced tokens to the syntactic analyzer.
RI.4	The lexical analyzer must be able of send the detected errors and warnings to the Error Management System.

Figure 5.7: Lexical analyser non functional requirements.

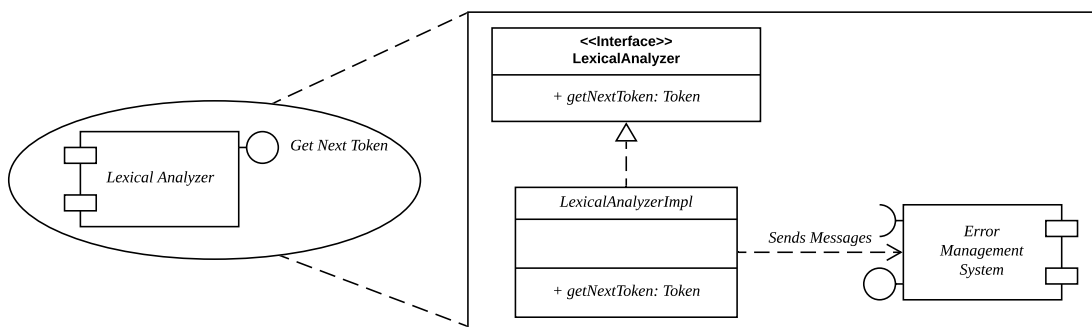


Figure 5.8: Lexical analyser component and class diagrams.

5.3 Syntactic Analyser

The parser may not be so necessary if we are looking to improve existing systems, but it is necessary to carry out the next step, semantic analysis. To others in this step you can also propose some improvement, although less. The following diagram shows the expected use cases for a system that wants to implement a syntactic validator.

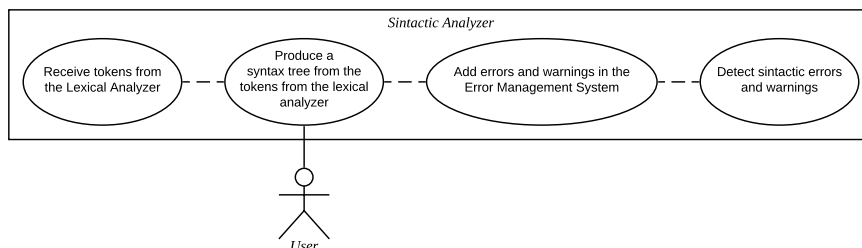


Figure 5.9: Syntactic analyser use cases.

Thus, from the use cases mentioned in the previous section we extract the following functional requirements.

ID	DESCRIPTION
RF.10	The syntactic analyzer must be able of analyze the tokens and produce a syntax tree.
RF.11	The syntactic analyzer must be able of detect syntactic errors and warnings.
RF.11.1	A syntactic error is produced when the input tokens does not match the ShEx micro Compact syntax specification.

Figure 5.10: Syntactic analyser functional requirements.

From the use cases we can also extract the following interface requirements.

ID	DESCRIPTION
RI.5	The syntactic analyzer must be able of send the produced sintax tree to the semantic analyzer.
RI.6	The syntactic analyzer must be able of send the detected errors and warnings to the Error Management System.

Figure 5.11: Syntactic analyser non functional requirements.

For the previous requirements we propose the following modulation for the syntactic analyser.

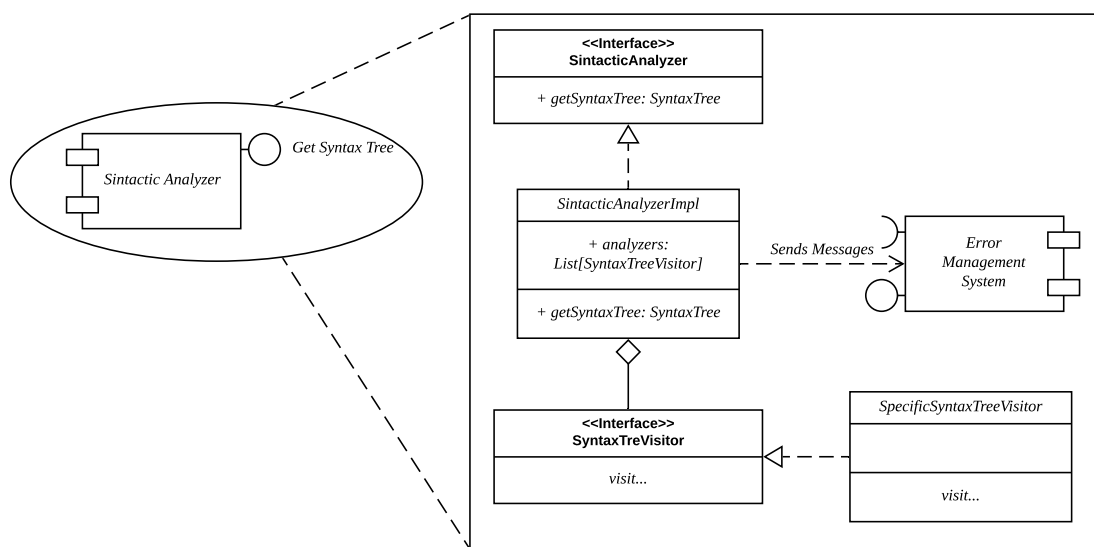


Figure 5.12: Syntactic analyser component and class diagrams.

5.4 Semantic Analyser

The semantic analyser is key in our architecture since most of the improvements that have to do with finding new types of errors can be identified through semantic validations. The following diagram shows the expected use cases for a system that wants to implement a semantic validator to solve the above problems.

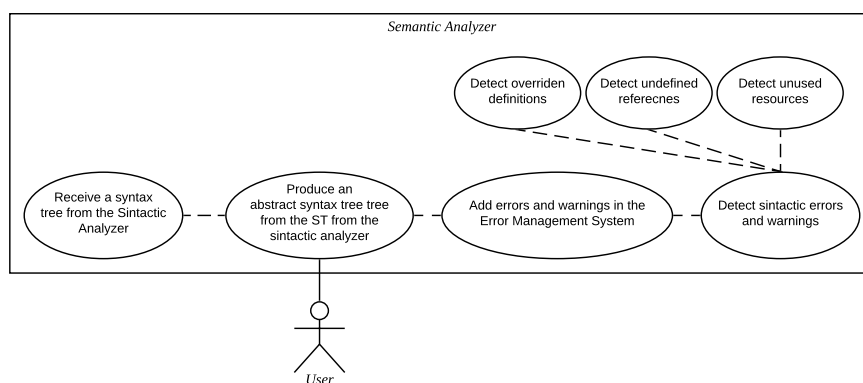


Figure 5.13: Semantic analyser use cases.

Thus, from the use cases mentioned in the previous section we extract the following functional requirements.

ID	DESCRIPTION
RF.12	The semantic analyzer must be able of analyze the syntax tree and produce an abstract syntax tree.
RF.13	The semantic analyzer must be able of detect semantic errors and warnings.
RF.13.1	A semantic error is produced when the conditions from section 4.4 are given.

Figure 5.14: Semantic analyser functional requirements.

From the use cases we can also extract the following interface requirements.

ID	DESCRIPTION
RI.7	The semantic analyzer must be able of send the detected errors and warnings to the Error Management System.

Figure 5.15: Semantic analyser non functional requirements.

For the previous requirements we propose the following model for the semantic analyser.

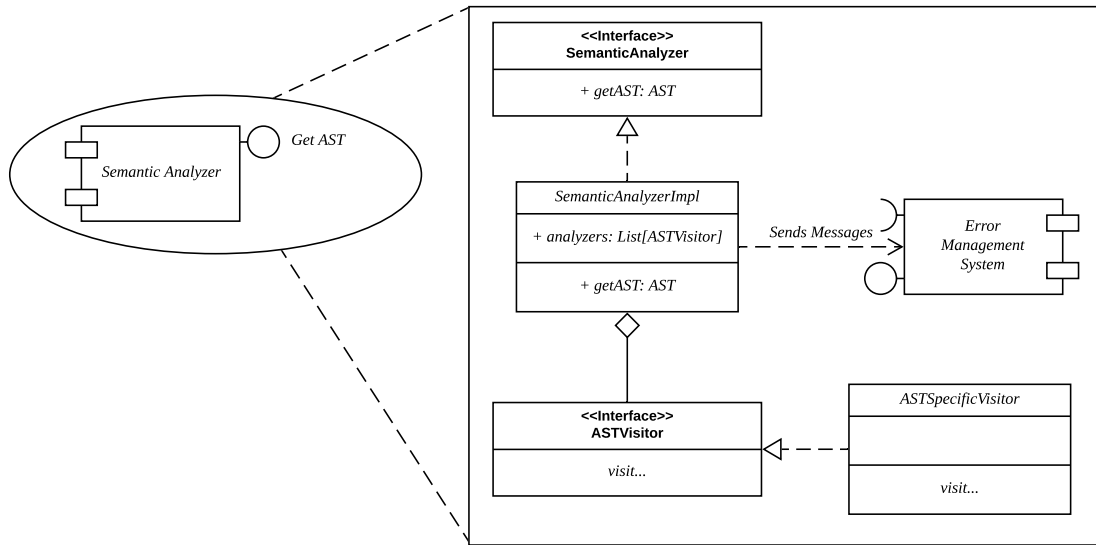


Figure 5.16: Semantic analyser component and class diagrams.

5.5 Full System Diagram

After analysing and designing each component now we offer a complete view of the entire integrated system. In addition you can see that a new component appears, the symbol table. This component can be any type of structure that fulfils the expected basic functions of a symbol table.

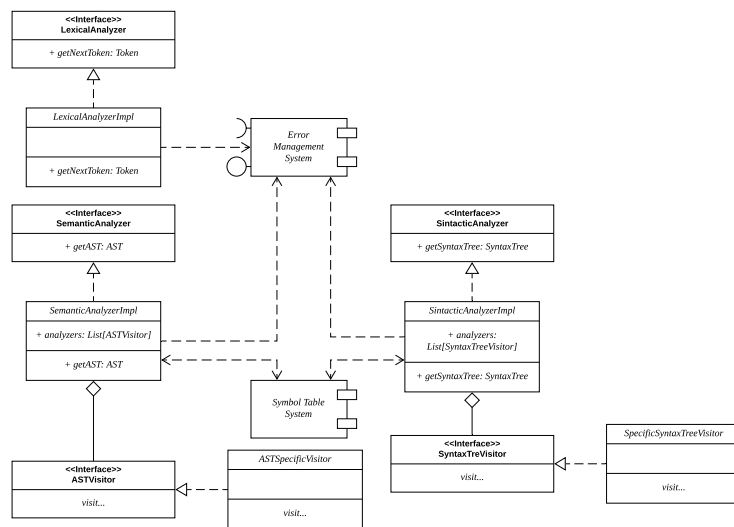


Figure 5.17: Complete system class diagram.

CHAPTER 6

Proposed Implementation

Once all the objectives and requirements to be achieved have been described, the different systems and techniques existing to achieve them have been studied, and their contributions and shortcomings have been evaluated, we will describe the proposed solution both in terms of design and possible implementation

6.1 Structure

The system is divided into components so that each component works on its input and produces its output. In this way, a parser is achieved that behaves like an API where each element can be called individually. [Figure 6.1](#) shows the different components of this analyser.

6.1.1 Parser

We define the parsing stage as the process that begins when we receive the source code that makes up the schema until the moment we produce a syntax tree. Therefore it includes the conversion to tokens by the lexer, the grouping of tokens in rules and later in a syntax tree

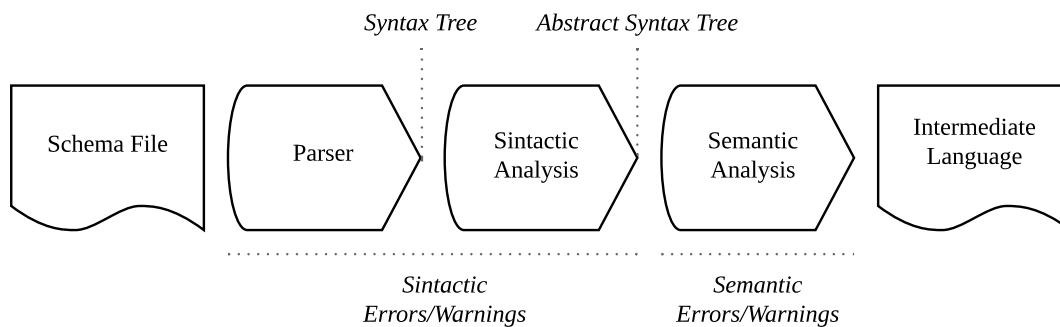


Figure 6.1: Syntactic and Semantic Analyser structure.

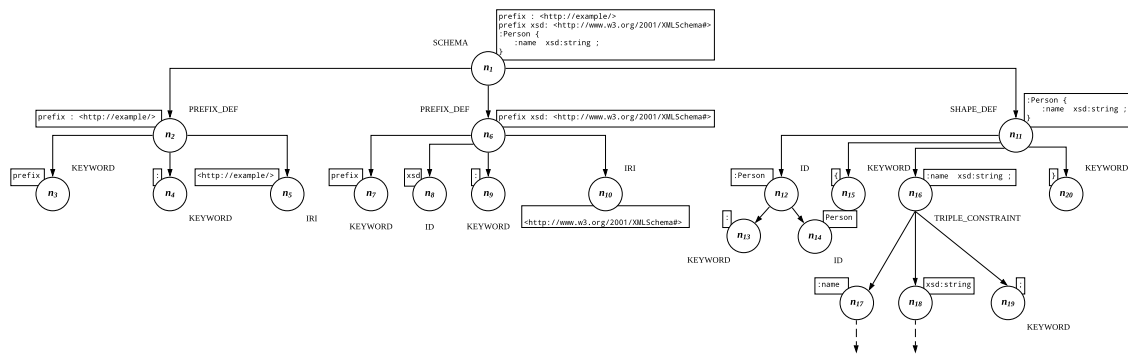


Figure 6.2: Syntax Tree twenty first nodes produced by the parser.

by the parser.

The general idea of this stage is that you take the source code as input and build a syntactic tree with all the possible information from the source code. This implies that the syntactic tree is not only made up of abstract grammar, but also of separators, braces and keywords. Figure 6.2 shows an example of the first 20 nodes generated by the parser. There we can see this composition of separators, keywords, braces and content.

Once we have the complete syntactic tree generated, we can go through it to carry out syntactic analysis on the different elements. For example, in the tree in Figure 6.2 we could implement a validator that in the event that the last triple constraint of a shape definition (node 16) did not have the semicolon termination keyword (node 19), it would generate a warning message to the user.

6.1.2 Syntactic Analyser

The Syntactic analyser is in charge of traversing the syntactic tree in order to search for possible patterns that the user has to be informed about. If none were found it would be understood that the syntactic tree is well formed and it will transform the Syntax Tree Figure 6.2 into an Abstract Syntax Tree Figure 6.3 (without the green and red relations).

For this, each node within our syntactic tree is aware of the context in which it is. Therefore, it is possible to access contextual information of a given node. For instance, we can access to a prefix definition (Figure 6.2 n_2) contextual information to check if it contains a label. It does not contain one. Or access the information of (node n_5) to get the node that defines its iri. Would be (node n_5). With questions like these, the syntactic tree can be analysed for patterns that represent warnings or errors.

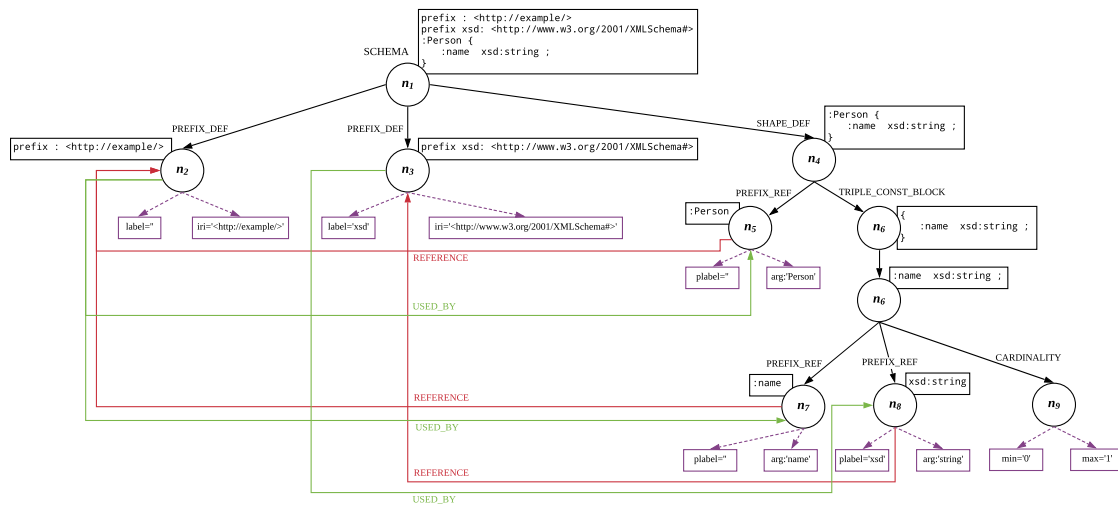


Figure 6.3: Abstract Syntax Tree produced after validation and transformations.

6.1.3 Semantic Analyser

The semantic analyser is responsible of building all the possible relations between the AST nodes, analyse and check that all those relations that must exist indeed exist. For this purpose as just seen we reduce our Syntax Tree to an Abstract Syntax Tree. Figure 6.3 Shows the resulting AST after the corresponding analysis and transformations, we call this graph the *Intermediate Language*.

Once we have the representation modelled and this representation is capable of expressing all the assumptions of our language, we can begin to apply validators on our structure. For example if we wanted to find broken references we could go to the nodes that are a reference to definitions like nodes $n_{5,7}$ and n_8 and check that there is indeed a valid reference for each of them.

Furthermore, we can even analyse how many times a definition is used by a reference so that we can launch messages warning the end user in some cases, such as when a prefix is not used.

6.2 Implementation

As proof of concept of the previous proposal we offer an implementation of the three components, the parser, the Syntactic analyser and the semantic parser. The implementation is defined in the same way as the structure, in three parts. We will now explore each of those parts and their responsibilities separately.

```
1  override def visitConstraint_triple_expr(...) {
2      if(/*No trailing semicolon*/)
3          //Warn user about this bad practice
4  }
```

Figure 6.4: Checker implementation for missing semicolons warning generation.

6.2.1 Parser

As previously discussed, the function of the parser is to extract a syntactic tree from the diagrams that we can analyse. For this purpose we decided to use the Antlr tool [28]. This tool is capable of generating Syntactic analysers from grammars defined in its own syntax. However, this tool is focused on completely processing the syntax tree and producing only the abstract syntax tree. Therefore we had to use a modification of the original ShEx micro Compact Syntax syntax so that Antlr would produce a tree with all the syntactic content. This also does offer the flexibility that in the future if we want to implement any additional syntactic validation we simply have to do it on the tree that the parser generates for us and not on the Antlr code.

6.2.2 Syntactic Analyser

The Syntactic analyser has the responsibility to validate that the parser produced syntax tree is correct and to build the abstract syntax tree as well. To do this, it uses the same mechanism. Through the visitor pattern we go through our syntax tree. Each implementation of this visitor has a purpose, for example an implementation can go through a few specific nodes to validate them syntactically while another can go through them in order to build the AST. [Figure 6.4](#) shows an example of how a Syntactic check is implemented.

The AST construction stage is very delicate since for each generated node we have to include as much context information as possible so that when an error is detected in the tree we can identify not only the cause but also the position, the origin, the rest of the affected nodes and therefore offer a content-rich error message. Regarding our implementation, for each node we save the following context information:

- **Source file path.** It represents the path to the source file where the node was generated.
- **Line.** The line in the source file where the node was generated.
- **Column.** The column in the source file where the node was generated.
- **Token interval.** The interval (*start, end position*) of tokens from the source file that generated the node.
- **Content.** The content of the node as plain text. [Figure 6.3](#) is very representative of

```

1 warning[W005]: missing semicolon
2 --> shape_with_warning_cause_semicolon.shexl:8:23
3 |
4 8 | :knows      @:User *
5 |           ^ semicolons are not compulsory in the last triple constraint,
6 |           but its usage its encouraged as otherwise your code wont be
7 |           following shape expressions specification.

```

Figure 6.5: Syntactic warning produced by the proposed Syntactic analyser.

<i>Property</i>	<i>Value Type</i>
<i>Source File</i>	<i>String</i>
<i>Line</i>	<i>Integer</i>
<i>Column</i>	<i>Integer</i>
<i>Token Interval</i>	<i>Interval<Integer></i>
<i>Content</i>	<i>String</i>
<i>Father</i>	<i>Node</i>
<i>Children</i>	<i>List<Node></i>

n_x

Figure 6.6: Common information stored at any AST node.

this.

- **Parent node.** A pointer to the parent node.
- **Children nodes.** A list of pointers to all the children nodes.

Figure 6.6 represents this information inside each node. Our default implementation only looks for the following extra syntactic pattern to the other implementations seen in Chapter 4: shape expressions whose last triple constraint does not contain the semicolon ending character. In case we find this pattern, we inform the event manager that a notice has been found that must be passed on to the user, Figure 6.5.

6.2.3 Semantic Analyser

Recall that the semantic analyser takes the generated AST, runs it in search of errors and transforms it in such a way that it emits a graph that corresponds to the intermediate language. We can separate semantic analysis into two phases, a first one in which we transform our syntactic tree, adding possible relationships. And a second phase in which we analyse existing and created relationships.

Tree transformations

In the case of our syntax the semantic relations that we find is the linking of a reference to its definition and the opposite direction to indicate that a definition is being referenced by a node. The transformations are listed below:

- **Linking prefix definition with prefix references.** Prefix references occurs when a node describes itself as the composition of a prefix and an argument. The idea is that the prefix substitute the IRI, but must be linked as any prefix reference needs to point to an existing definition.
- **Linking base definition with base references.** Some nodes are defined as relative IRIs to the base definition and therefore need to be linking to them in order to be able to get that base IRI.
- **Linking shape definition with a shape reference.** Shape definitions can be used at the `start` definition to point the default shape or as type constraints in the triple constraints. At any of those points shape references must exist within the scope of an schema.

Tree relations analysis

For this purpose, the semantic analyser defines the visitor pattern on the nodes of the abstract syntax tree so that each of the different analysis is done with a tree visiting implementation.

6.3 Tests

Different types of tests were used to validate the implementation. Of course, unitary, integration, regression. However, the most important in here are the dynamic test cases. In this way it was possible to test the system just by describing test cases using files. These files represent the input of the test system. And according to the file name, it is automatically classified as positive or negative test. A battery of tests defined by the people who carried out the ShEx specification have been used to tests the system. In our case with 45 different test scenarios. In addition, to ensure compatibility with all platforms, continuous integration with GitHub has been made over Windows, Linux and Mac OS. A coverage of the 70% of the code is achieved. See [Chapter C](#) to see the CI configuration.

6.4 Syntactic and Semantic Error and Warnings Detected

With the solution proposed in the previous section, our system is capable of detecting and reporting multiple syntactic and / or semantic errors. In this section we will analyse the rules that generate each type of event and the different error messages produced for each one of them.

```
1 error[E007]: prefix not defined
2 --> shape_with_error_cause_pref_not_defined.shex:17:3
3   |
4 17 | non_existing:label xsd:string +;
5   | ~ the prefix 'non_existing' has not been defined
```

Figure 6.7: Semantic error produced by an undefined prefix.

6.4.1 Not trailing semicolon at last triple constraint

To detect when the semicolon is missing in the last triple constraint of a shape definition, the rule used is very simple. Find the last node in the triple constraints list of a shape definition. And once this node is found, it is searched whether or not it contains the final token character that corresponds to a semicolon. If it does not have it, a warning message is generated, indicating the position through file, line, column and context, which is sent to the compilation event manager, which in turn gives the corresponding format to print the message. [Figure 6.5](#) shows an example of this message.

6.4.2 Prefix not defined

These types of events happen when we use a reference to a prefix and this has not been defined in the scope of the schema. In the event that this happens we have an error that we cannot recover from since we cannot associate the reference to anything.

In order to detect this assumption, all the prefix definitions have to be traversed previously and for each one of them, a record will have to be created in a symbol table where it is indexed by the label and a reference to the definition node is added. All types of type reference to prefix can then be accessed and for each one it is verified that the label exists in the symbol table and then a pointer to the corresponding definition node is added to the reference node. If, on the other hand, a definition cannot be found in the symbol table, then an error message like [Figure 6.7](#) is created.

For example in the [Figure 6.3](#) the red lines would be the transformations done to the original AST to add the pointers to the reference nodes that point to the definition nodes.

6.4.3 Shape not defined

In the same way as the previous case, an undefined shape error occurs in the case that there is a reference to a shape expression that is not defined in the scope of the schema.

For this, all the definitions of shape expressions of our schema must have been previously identified and indexed in a symbol table where the key is the name and the value a reference to the node of the definition. Once the definitions of shape expressions have been identified, we only have to go through those nodes of type reference to shape expression and look for a

```

1 error[E008]: shape not defined
2 --> shape_with_error_cause_shape_not_defined.shex:16:13
3 |
4 16 | @existing_prefix:Not_Existing_Shape
5 | | ^ the shape 'Not_Existing_Shape' has not been defined
6 | | in the scope of the prefix 'existing_prefix'

```

Figure 6.8: Semantic error produced by an undefined shape.

```

1 error[E003]: attempt to override an already defined prefix
2 --> shape_with_error_cause_prefix_override.shex:15:0
3 |
4 15 | PREFIX foaf: <hppt://another/value>
5 | | ^ this prefix definition overrides the previous one (9:0) with
6 | | value <http://xmlns.com/foaf/0.1/>

```

Figure 6.9: Semantic error produced by a prefix override.

definition of a shape expression with the corresponding label within the scope of the prefix specified in the reference. If it exists, a reference is added to the type reference to shape that points to the corresponding definition. Otherwise, an error message like [Figure 6.8](#) is generated.

6.4.4 Prefix overridden

We say that a prefix is overwritten when we come across a second prefix definition that tries to assign any value to a prefix that had already been defined previously.

For this, during the identification of prefixes, every time we find a prefix type node we try to add a record to our symbol table. In this entry, the key will be the prefix label. If there is already an entry in the symbol table with the same tag, then we would be facing a prefix override. So instead of taking the action we would throw an error message like [Figure 6.9](#).

6.4.5 Shape overridden

The case of a shape expression overwriting is slightly less trivial in that a shape is identified as the union of an existing prefix and a unique identifier within the ambit of that prefix. Therefore, the way of acting will be (assuming that the prefix exists, if it would not be another error) check if a shape definition with the same identifier already exists within the scope of the indicated prefix. If it exists we will throw an error like the one from [Figure 6.10](#). If not, we will add a record to the indicated prefix scope with the corresponding information from the shape definition.


```

1 error[E004]: attempt to override an already defined shape
2 --> shape_with_error_cause_shape_override.shex:40:0
3   |
4 40 | :Q3559 {
5   |   ^ this shape definition overrides the previous one (17:0)
6 41 |   schema:name xsd:string ;
7 42 | }

```

Figure 6.10: Semantic error produced by a shape override.

```

1 warning[W001]: prefix definition not used
2 --> shape_with_warning_cause_prefix_never_used.shex:8:12
3   |
4 8  | PREFIX owl: <http://www.w3.org/2002/07/owl#>
5   |   ^ the prefix 'owl' definition is never used

```

Figure 6.11: Semantic warning produced by a prefix never used.

6.4.6 Unused prefix definition

One of the small optimizations that our semantic solution includes is the early detection of resources not defined as prefixes. In addition, it is a use case of semantic statistics generated by our proposed solution. In this specific case, what is checked is the number of resources that use a definition. For this, the symbol table is consulted since this is the one that stores this information. It corresponds to the relationships in green in [Figure 6.3](#).

In the event that a prefix definition has zero resources that use it, the prefix is not used and therefore it can be removed without problem since it only takes up space. To warn the user of this, a warning like [Figure 6.11](#) is generated

6.4.7 Base set but not used

Another case in which the early detection of unused resources is used is with the definition of the base. If for some reason a user assigns a value to the base but never uses it, a warning like [Figure 6.12](#) is generated.

```

1 warning[W002]: base has been set but not used
2 --> shape_with_warning_cause_base_set_but_never_used.shex:17:5
3   |
4 17 | BASE <http://a/base/not/used/value>
5   |   ^ the base '<http://a/base/not/used/value>' definition is
6   |     set but not used

```

Figure 6.12: Semantic warning produced by a base set but never used.

Part II

Translating ShEx Schemas to Object Domain Models

CHAPTER 7

Object Domain Model Translation Problem

The ODMTP (*Object Domain Model Translation Problem*), when talking about Shape Expressions, is the aim to transform existing schemas, that already represent domain models, into object domain models. I.E, translate the ShEx schemas to objects coded in some Object Oriented Language. [Figure 7.1](#) represents this aim. The problem is to convert the *Source* in to the *Target* (*shex* \rightarrow *object oriented language*).

Person Schema (Source)	Person Java Object (Target)
1 # Prefixes...	1 // Imports...
2 :Person {	2 public class Person {
3 :name xsd:string ;	3 private String name;
4 :knows @:Person *	4 private List<Person> knows;
5 }	5 // Constructor...
	6 // Getters and Setters...
	7 }

Figure 7.1: Schema modeling a Person in ShExC syntax to the left. And the expected translated code in Java to the right.

This problem, with the previous example [Figure 7.1](#), may seem simple to solve, however, before proposing a solution, we need to explore if everything that can be expressed with ShEx can be expressed in object-oriented languages.

To answer this question, we will reduce our problem by using the micro ShEx syntax and Plain Objects (*PO*) [29] as a generalization of all the programming languages that support the object orientated paradigm. Therefore our study will focus on finding out if we can express in plain objects everything we can express in the ShEx micro syntax. [Formalization 7.1](#) illustrates this question where $e(x)$ measures the expressivity [30] of x .

$$e(\text{shex micro syntax}) \leq e(\text{plain objects}) \quad (7.1)$$

So, the first step will be to measure the expressivities of both the ShEx micro syntax and the Plain Objects to later compare them.

```

1 schema          ::= definition+
2 definition      ::= prefixDef | baseDef | startDef | shapeDef
3 prefixDef      ::= ID IRI
4 baseDef        ::= IRI
5 startDef       ::= SHAPE_REF
6 shapeDef       ::= IRI_REF tripleExpression+
7 tripleExpression ::= IRI_REF constraint CARDINALITY
8 constraint     ::= IRI_REF | SHAPE_REF | "IRI" | "BNODE" |
9                 "NONLITERAL" | "LITERAL"

```

Figure 7.2: ShEx Micro Abstract Grammar.

7.1 Shape Expressions Expressivity

To explore the expressiveness of ShEx micro Compact Syntax we have to look at the abstract grammar (Figure 7.2) of the syntax. In it we will find what we can and what we cannot express. For example we can deduce that an schema is a set of shapes where each is defined as an identifier and a set of triple expressions. Figure 7.3 shows an example of a shape expression coded on its micro compact syntax that defines two properties for the object `:Person`. In that shape expression we can see that we have a property that represents the name with type string and the default cardinality (1). And a second property `knows` whose type is a reference to another person and has `0...*` cardinality so it represents a list of people you know.

However, only with the grammar it would be very difficult for us to compare with other expressiveness. For this purpose we will obtain the formalization based on the [31] formalization on RDF graphs.

Let U be the set of URI-s, B the set of blanks and L the set of literals. Let us also define sets, $P = U$, $T_{rdf} = U \cup B \cup L$ and $C = \{(n, m) \mid n, m \in \mathbb{N}, n \leq m\}$. Then a triple expression is defined as

$$(p, t_{rdf}, c) \in P \times T_{rdf} \times C, \quad (7.2)$$

where p represents the property, t the node constraint and c the cardinality. A shape

$$s \subseteq P \times T_{rdf} \times C \quad (7.3)$$

is a set of triple expressions which implies that an schema

$$S = \{s \mid s \subseteq P \times T_{rdf} \times C\} \quad (7.4)$$

is a set of shapes. Thus, the expressivity of a shape expressions schema will be given by $P \times T_{rdf} \times C$. Therefore, $e(\text{shex micro syntax}) = P \times T_{rdf} \times C$.

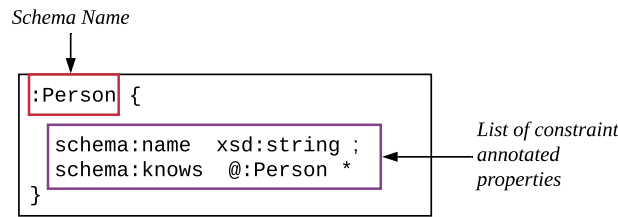


Figure 7.3: Shape expression modeling the properties of a Person.

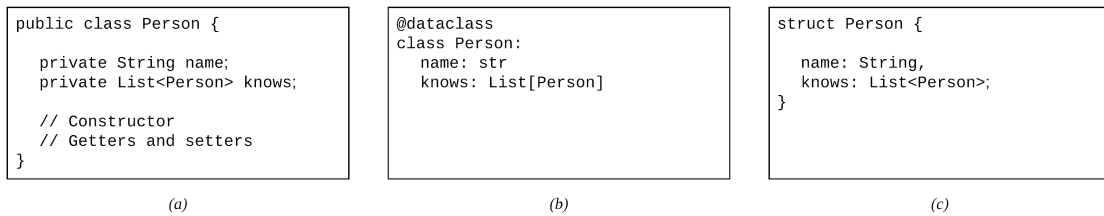


Figure 7.4: Java, Python and Rust codings of Person object. *a* corresponds to Java, *b* corresponds to Python and *c* corresponds to Rust.

7.2 Plain Objects Expressivity

Plain objects can be coded in any object oriented programming language, or at least in any language that supports this paradigm. First we will explore how plain objects are generally coded, then how the language increases or decreases the expressivity and finally we will generalize the core concepts that can be expressed by any plain object codification.

7.2.1 Plain Objects Structure

From the existing programming languages we can infer the general structure of plain objects. For this purpose we take the PYPL Index (*PopularitY of Programming Language*)¹ from June 2020 and take the 2 most used programming languages that support the object oriented paradigm, those would be Java and Python. And then, just to enlarge the scope we will take Rust because it is a new programming language that includes lots of features.

Figure 7.4 shows three models that correspond to the codification of the Person schema from Figure 7.1. For example if we analyze the Java fragment, that seems to be the most complex one out of the three fragments we can see in Figure 7.5 that it is composed by the *Schema Name*, the *List of Type Annotated Properties* and some *Language Specific Code*. This

¹<http://pypl.github.io/PYPL.html>

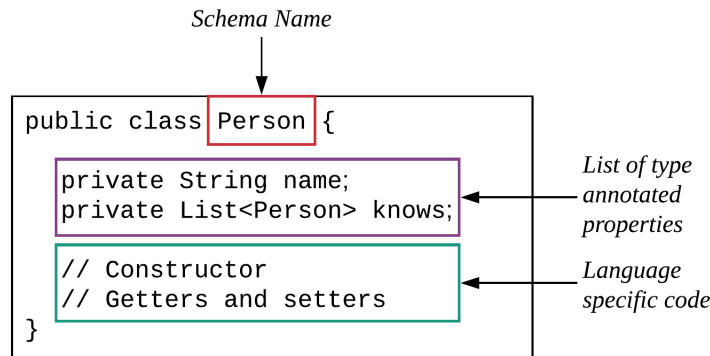


Figure 7.5: Java plain object decomposition.

correlates to the other two programming languages as they also contain this three elements.

It is important to note that although the composition of the property identifiers may vary a little in each programming language, the type system is specific and sure to change in each language. This is why we must explore to what extent it affects the type system of each language. And if, therefore, it can be generalized.

7.2.2 Plain Objects Formalization

In order to compare the expressiveness of plain objects with other systems, we will carry out a formalization based on their structure and content. Let N be the set of all possible variable identifiers in a programming language and T_{pl} the set of all possible types in a programming language. Then a type annotated property is defined as

$$(n, t_{pl}) \in N \times T_{pl}, \quad (7.5)$$

where n is the identifier of the property and t_{pl} the specific programming language type. That implies that a plain object

$$c \subseteq N \times T_{pl}. \quad (7.6)$$

is a set of type annotated properties. Thus, an object domain model

$$M = \{c \mid c \subseteq N \times T_{pl}\}. \quad (7.7)$$

is a set of plain objects. Therefore, $e(\text{plain objects}) = N \times T_{pl}$.

7.2.3 Plain Objects Language Expressivity Dependence

In [Figure 7.4](#) we can see that all of three languages use similar types to represent the Person model. But with just one example we cannot generalize that the language does not affect the expressivity of the plain objects. In order to test that condition and prove that the language affects or does not affect the expressivity of plain objects we will need first to find two *type-independent languages*.

Definition 7.1 (Type-independent languages). *Two languages L_1 and L_2 are type-independent if and only if one of the languages contains a type that cannot be represented by means of a linear combination of any other type of the other language.*

For example, lets take Java L_1 and Rust L_2 , examples (a) and (c) from [Figure 7.4](#). Rust contains the type *Either* $\langle A, B \rangle$, this type allows the type A or B and when accessed is not an *Either* is either A or B . In Java there is no *Either* type, and someone can say that we could achieve a similar type by using inheritance and classes composition. But at the end when accessed the type would be the type of the upper class. **Therefore Java and Rust are type-independent languages.**

Now in order to see if the expressivity depends on the types of a language let's assign values to Java and Rust by using the same *Either* $\langle A, B \rangle$ type. As can be see in [Figure 7.6](#) Java does not allow to express the same as we are expressing in Rust in this example. And therefore, we can conclude that the expressivity of plain object is strongly related to the build-in types that the programming language in which they are coded provides.

Person Rust Struct	Person Java Object
<pre> 6 struct Person { 7 name: String, 8 knows: List<Person>, 9 owningPet: Either<Dog,Cat>, 10 }</pre>	<pre> 8 // Imports... 9 public class Person { 10 private String name; 11 private List<Person> knows; 12 private Pet owningPet; 13 // Constructor... 14 // Getters and Setters... 15 }</pre>

Figure 7.6: Rust struct modeling a **Person** to the left. And the most similar approximation in Java to te right. In the Java approximation the Pet class is an interface that it is inherited by the Cat and Dog classes, that way we allow to store in the variable `owningPet` values of type Cat and Dog.

7.2.4 Plain Objects Expressivity Generalization

In order to obtain a generalization of the plain objects represented by means of object-oriented programming languages, we will base ourselves on [Formalization 7.5](#) where we defined the composition of a plain object, in this way the generalization would be as indicated in

```
1 plain object ::= (ID type)+
```

Figure 7.7: Plain Objects Partial Generalization.

```
1 plain object ::= (ID type)+
2 type        ::= REAL | LIST[type] | STRING | BOOLEAN | ID
```

Figure 7.8: Plain Objects Complete Generalization.

[Figure 7.7](#). As can be seen this generalization is not complete as it does not include the production for the `type`. This is because we have not generalized the type system of the object oriented programming languages yet.

However, and motivated by not to over-extend the scope of this work, instead of extracting a generalization for all the possible types that can be used in each object-oriented programming language, we will try to create this abstraction projecting the most common types used by XML Schema (xsd) [32]. The main reason, is that in RDF, and therefore in ShEx, xsd is the most widely used type system and the standard of w3c. This leads us to the generalization from [Figure 7.8](#) where we re-use the xsd types and add the ID that actually represents compound types, that is types that are in fact plain objects.

7.3 Shape Expressions and Plain Objects Expressivity Comparison

Previous section cover the expressivity of Shape Expressions and Plain Objects. In this section we compare both expressivities and expose if both expressivities are compatible or not. [Formalization 7.4](#) showed that the expressiveness of the schemes depends on $P \times T_{rdf} \times C$. Meanwhile [Formalization 7.7](#) showed that the expressivity of object domain models depends on $P \times T_{pl}$. In [Section 7.2](#) we restrict the types that a plain object property might have. Let T_g be the set of all allowed types in a plain object $\{Real, List_{T_g}, String, Boolean, Reference\}$. Then, $e(\text{plain objects}) = N \times T_g$. We will compare now if $P \times T_{rdf} \times C = N \times T_g$. For that purpose we will compare each space separately to see if they represent the same.

$$\left. \begin{array}{l} P \rightarrow \text{All the URIs.} \\ N \rightarrow \text{All the identifiers.} \end{array} \right\} N \subseteq P \quad \left. \begin{array}{l} T_{rdf} \rightarrow \text{All RDF types.} \\ T_g \rightarrow \text{All allowed plain objects types.} \end{array} \right\} T_g \subseteq T_{rdf} \quad (7.8)$$

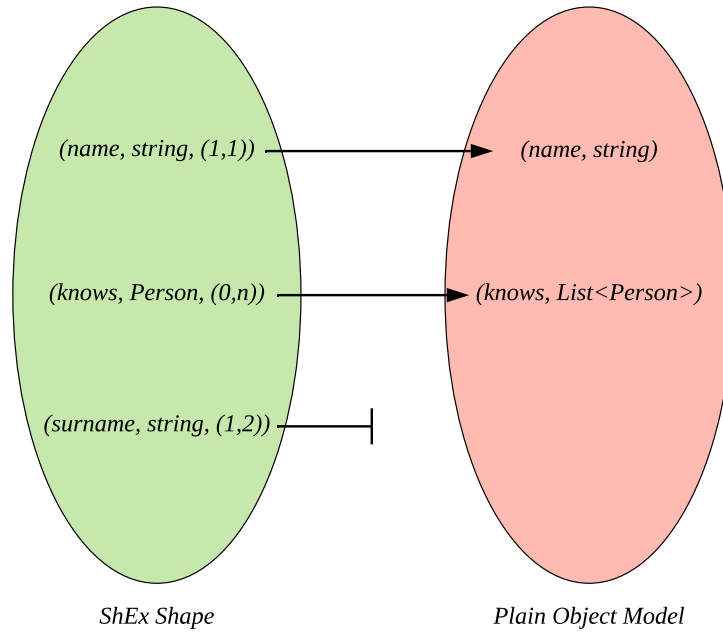


Figure 7.9: Mapping function from ShEx to Plain Object.

Here we can see that the C has no space to compare with as s does not include any information about this. In s instead of using a cardinality there is a list type that represents the cardinality $(0, \infty)$.

From here we see that $e(\text{shex micro syntax}) \not\subseteq e(\text{plain objects})$. So that answers the question. As the expressivity of Shape Expressions Micro Compact Syntax is greater than the expressivity of the defined Object Domain Models we cannot transform all the existing schemas from S to plain objects from M . That can be easily proven also by assigning values to S and trying to map them in to M . [Formalization 7.9](#) and [Figure 7.9](#) illustrate this.

$$\begin{array}{ll}
 (\text{name}, \text{string}, (1, 1)) \rightarrow & (\text{name}, \text{string}) \\
 (\text{knows}, \text{Person}, (0, \infty)) \rightarrow & (\text{knows}, \text{List}[\text{Person}]) \\
 (\text{surname}, \text{string}, (1, 2)) \rightarrow & \times \\
 (\text{enrolled}, \text{Person}, (2, \infty)) \rightarrow & \times \\
 (\text{length}, \text{inches}, (1, 1)) \rightarrow & \times
 \end{array} \tag{7.9}$$

[Figure 7.9](#) also illustrates perfectly that there exists some cases where the transformation can take place. Now we will focus on finding those cases. From [Formalization 7.8](#) we know that both $N \subseteq P$ and $T_g \subseteq T_{rdf}$ and previously we saw that if $c \in (1, 1), (0, \infty)$ we can represent

that by means of the type List. Therefore

$$\forall (p, t, c) \in N \times T_g \times \{(1, 1), (0, \infty)\} \quad (7.10)$$

both systems share the same expressivity. Which implies that a shape' $s' \subseteq N \times T_g \times \{(1, 1), (0, \infty)\}$ is a set of triple constraints that can be represented as type annotated properties. And therefore $S' = \{s \mid s' \subseteq N \times T_g \times \{(1, 1), (0, \infty)\}\}$ is equivalent to $M = \{c \mid c \subseteq N \times T_g\}$.

Proposed Translator

This chapter focuses on proposing a solution to ODMTP. First formalizing the solution. And then modelling it with elements of software engineering such as use cases and requirements.

8.1 Translator Formalization

As a solution to the previous chapter, this section focuses on proposing a application $f : S' \rightarrow M$ such that applied on a schema, results in a domain model based on plain objects.

Lets define $f(S') = \begin{bmatrix} f'(s'_1) \\ f'(s'_2) \\ \vdots \\ f'(s'_n) \end{bmatrix}$ and $f'(s'_i) = \begin{bmatrix} f''(e'_1) \\ f''(e'_2) \\ \vdots \\ f''(e'_n) \end{bmatrix}$. Then $f''(e'_i)$ is the application that

maps a triple expression e' from $N \times T_g \times \{(1, 1), (0, \infty)\}$ to $N \times T_g$. To find such a function we will use the knowledge that we already have. We know that p has a direct mapping as it belongs to N , T_g maps to T_g if the cardinality value is $(1, 1)$ or $(1, \infty)$. And the cardinality is aggregated to the type so its not needed to map it. Then we define the application $f''(e')$ as $f : (p, t, c) \in N \times T_g \times \{(1, 1), (0, \infty)\} \rightarrow (n, t) \in N \times T_g$ and therefore,

$$f''(e'_i) \begin{cases} (p, Proj_{t_g} lst) & \text{if } c = (1, 1) \\ (p, List[Proj_{t_g} lst]) & \text{if } c = (0, \infty) \end{cases}. \quad (8.1)$$

This application's function is to transform a triple expression into an annotated type property. Where the $Proj_{t_g} lst$ represents the projection of the generic type from the abstraction of languages of representation of plain objects on to the language specific type. [Figure 8.1](#) illustrates how the same input can lead to multiple types due to the specific translators, that perform the $Proj_{t_g} lst$ operation.

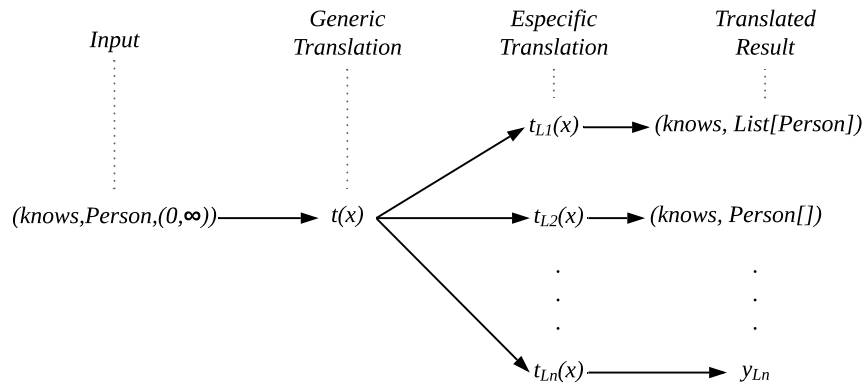


Figure 8.1: Different target types generated by specific translators.

8.2 Translator modelling

At this point we already have an abstraction of our system ready. We know that you must implement the transformation function previously explained. Now we will lower our abstraction one level. For this we will model our system by means of software engineering techniques such as use cases, requirements or class diagrams. First, we will use the use case method to find the necessary functionality of our system.

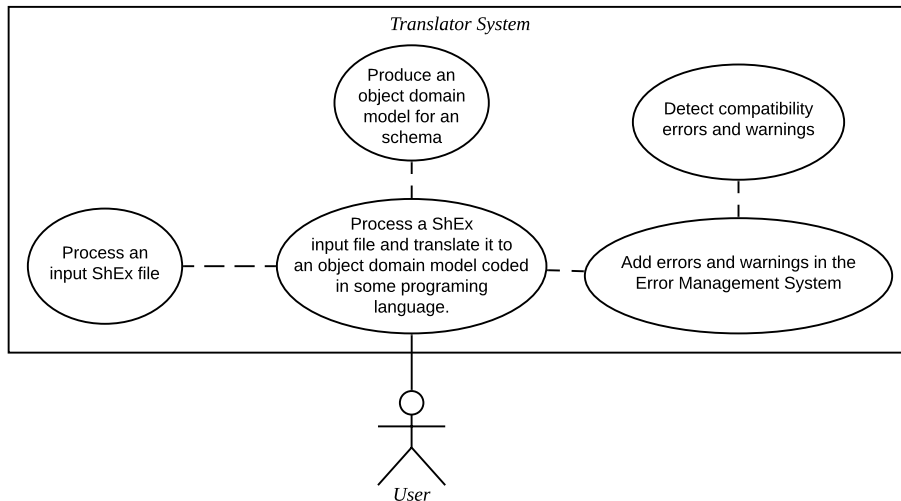


Figure 8.2: Translator use cases.

Figure 8.2 shows us that in our system we will, of course, have the functionality to translate ShEx schemas to domain models based on plain objects. But we can also see that the entry may be wrong. This implies that there must be some kind of input validation. In addition,

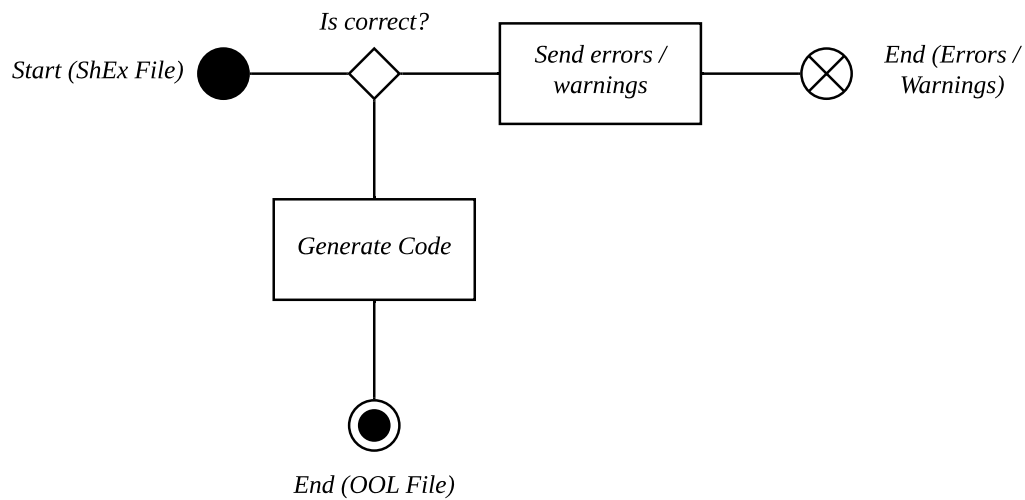


Figure 8.3: Translator high level flowchart.

an error management system is also necessary. [Figure 8.3](#) illustrates a high level view of the flowchart that the translator follows.

Also from the developed use cases we can extract the list of functional and non-functional requirements (external interfaces) that our system must support.

ID	DESCRIPTION
RF.14	The translator system must be able to translate an input ShEx file in an object domain model coded in some object oriented programming language.
RF.14.1	If the input file cannot be translated due to incompatibilities with the target language the translator system will add an error/warning to the Error Management System.

Figure 8.4: Translator functional requirements.

ID	DESCRIPTION
RI.8	The translator system will communicate with the Error Management System.

Figure 8.5: Translator non functional requirements.

Thus, for the previous use cases and requirements the implementation abstract diagram will be.

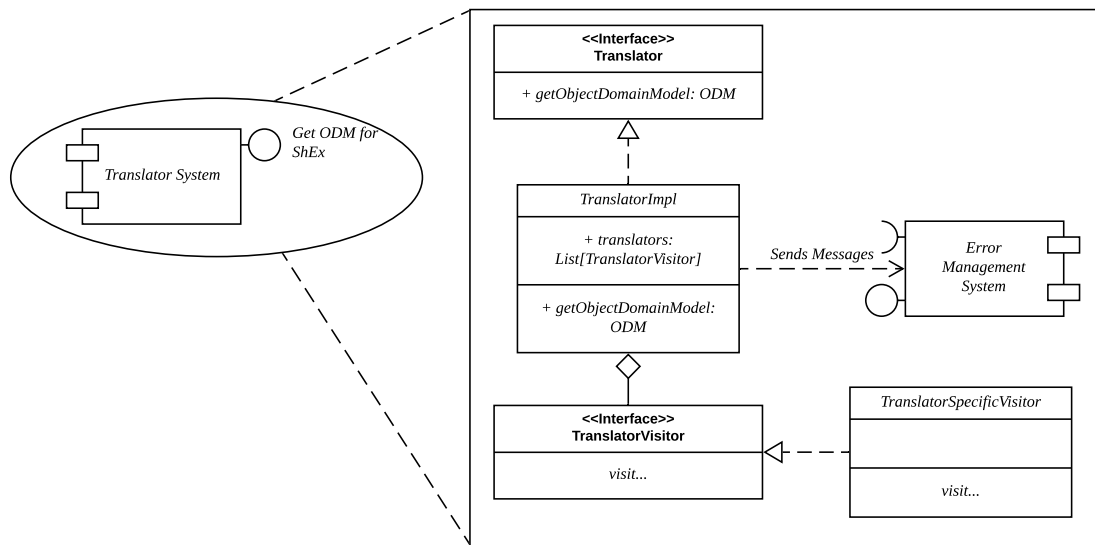


Figure 8.6: Translator component and class diagrams.

CHAPTER 9

Proposed Implementation

Our solution is based on a code translator. In the end, a translator is still a type of compiler [Figure 2.8](#), where we have the analysis and synthesis phase. The analysis phase focuses on verifying that the input is correct and on making the necessary transformations. While the synthesis starts from a high quality structure and performs the appropriate transformations to reach the target representation. In the case of our solution we have a difference and that is that we do not have a single target but multiple ones ([Figure 9.1](#)).

In addition to this, in [Chapter 6](#), we have already developed a system capable of analysing, validating and transforming source code so that we obtained an intermediate language made up of a high-quality graph. Therefore in our translator we will reuse the lexical, syntactic and semantic analyser from [Chapter 6](#), that makes the front end of our translator. And then, what we will truly develop in this section, is the back-end of the translator.

9.1 Translator Back-end

In our solution, the back-end of the translator, also called the synthesis phase, fulfils two main functions. On one hand, it analyses the intermediate language in search of some specific

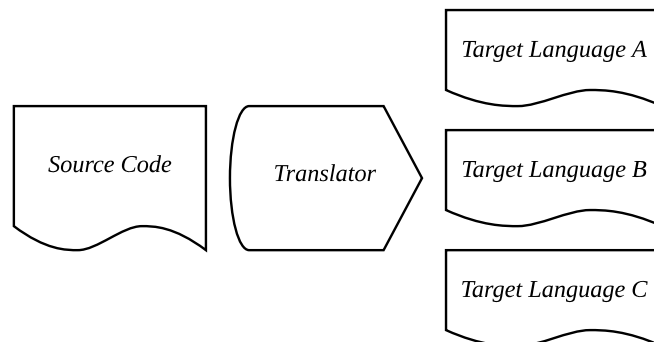


Figure 9.1: Translator generic structure.

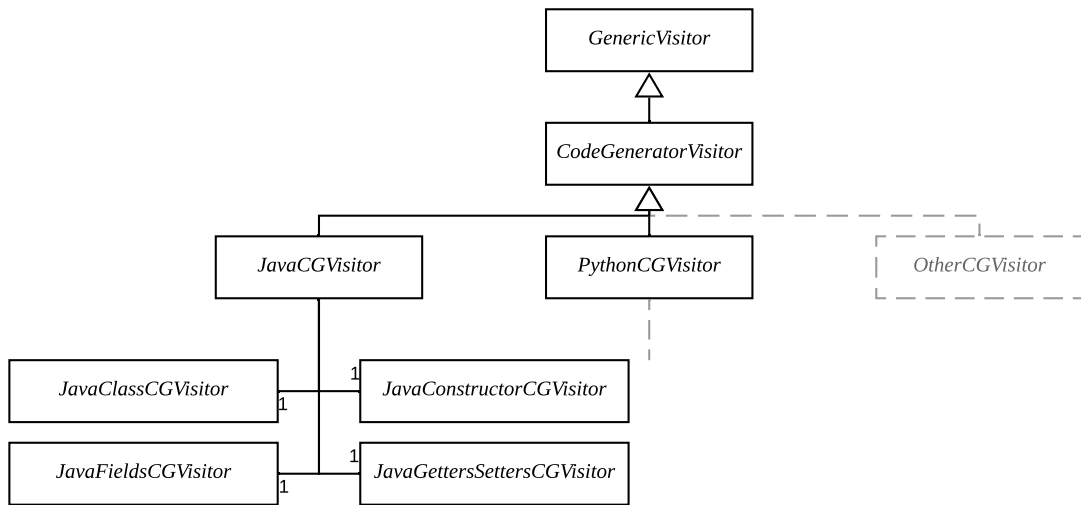


Figure 9.2: Class diagram of the code generation visitors.

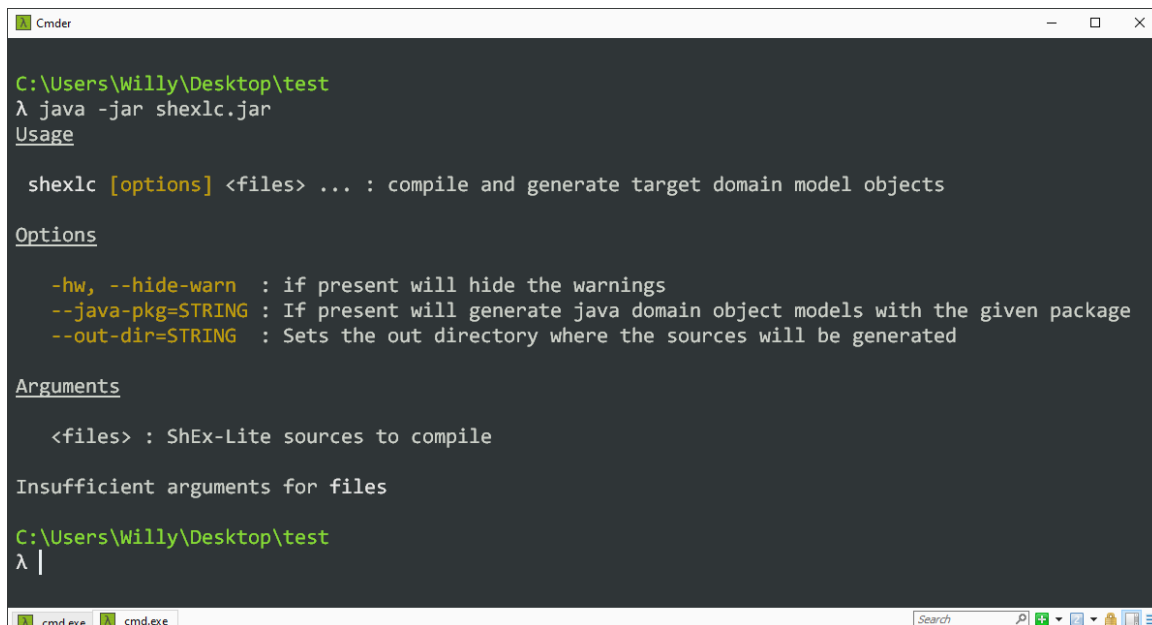
incompatibility with the target language. And on the other hand, it generates the specific code for the target language.

To perform the analysis of the intermediate language and through the Visitor pattern, a graph path is implemented that validates that no node or set of nodes violate the restrictions obtained in [Formalization 7.10](#). The visitor that is implemented is completely reused from the one developed for the Syntax or Semantic analysis in [Chapter 6](#).

For code generation, it is proposed to carry out an implementation of the Visitor pattern for each of the specific code generators, each one of the specific implementations will perform the transformation function described in [Formalization 8.1](#), without prejudice to the fact that each specific code generator may have more implementations of the visitor associated. [Figure 9.2](#) illustrates this situation where the Java code generation visitor actually contains four more visitors one for each language specific level of the plain objects.

And as proof of concept of the structure proposed in the previous section, we implemented a system that starts from the one developed in [Section 6.2](#) and is capable of generating code from schemas, checking that they are valid. This solution follows structure developed in this system and more precisely [Figure 9.2](#). Moreover this developed system offers a CLI tool ([Figure 9.3](#)). In this tool the users can define multiple options as `-java-pkg=STRING` which if present will trigger the java code generation and will generate the target object in the specified package.

For example, for the input `java -jar shexlc.jar -java-pkg=demo person.shexl` where the `person.shexl` file corresponds to the schema defined at [Figure 7.1](#) ShEx-Lite would



```
C:\Users\Willy\Desktop\test
λ java -jar shexlc.jar
Usage
shexlc [options] <files> ... : compile and generate target domain model objects

Options
-hw, --hide-warn : if present will hide the warnings
--java-pkg=STRING : If present will generate java domain object models with the given package
--out-dir=STRING : Sets the out directory where the sources will be generated

Arguments
<files> : ShEx-Lite sources to compile

Insufficient arguments for files

C:\Users\Willy\Desktop\test
λ |
```

Figure 9.3: CLI menu of ShEx-Lite CLI tool.

generate a single java class with the code that appears at the `Person.java` file, also in [Figure 7.1](#). This implemented system will be used for evaluating the proposed solution.

9.2 Tests

In order to validate the translator, it is no longer possible to rely on the tests previously defined by the ShEx specification. Therefore, synthetic tests have been designed for this purpose. These synthetic tests prove that each extreme case of the code generation system works. In addition, it is also proved that for each specific language of the code generation the type projection is correct. In addition, all these tests are run continuously on GitHub on Windows, Linux and Mac OS platforms. See [Chapter C](#) to see the CI configuration.

9.3 Generated Objects

In this section we will give real examples of use cases in which the proposed system has been used to generate objects, in addition we will study the objects in order to estimate their quality.

9.3.1 Real (Hercules ASIO European Project)

In the framework of the European project Hercules ASIO, financed with FIVE MILLION FOUR HUNDRED SIXTY-TWO THOUSAND SIX HUNDRED euros, the system described is used to link two parts of the architecture, the ontological infrastructure and the semantic

architecture. The Hercules project tries to find a solution based on linked data to manage the research framework in Spanish universities. Some examples of the objects generated in this project are [Figure 9.4](#) and [Figure 9.5](#).

University.shexl	University.java
<pre> 0 # Prefixes... 1 asio:University { 2 rdfs:name xsd:string ; 3 rdfs:county xsd:string ; 4 rdfs:location xsd:String ; 5 asio:hasRector 6 @asio:UniversityStaff ; 7 ... 8 }</pre>	<pre> 0 // Imports... 1 public class University { 2 private String name ; 3 private String country ; 4 private String location ; 5 private asio.UniversityStaff hasRector ; 6 ... 7 // Constructor... 8 // Getters and Setters... 9 }</pre>

Figure 9.4: Schema modelling a University in shexl syntax to the left. And the ShEx-Lite generated code in Java to the right.

Researcher.shexl	Researcher.java
<pre> 0 # Prefixes... 1 asio:Researcher { 2 rdfs:name xsd:string ; 3 rdfs:surname xsd:string ; 4 rdfs:id xsd:integer ; 5 rdfs:orcid xsd:string ; 6 rdfs:publications 7 @asio:AcademicPublication * ; 8 ... 9 }</pre>	<pre> 0 // Imports... 1 public class University { 2 private String name ; 3 private String surname ; 4 private int id ; 5 private String orcid ; 6 private List<asio.AcademicPublication> 7 publications ; 8 ... 9 // Constructor... 10 // Getters and Setters... 11 }</pre>

Figure 9.5: Schema modelling a Researcher in shexl syntax to the left. And the ShEx-Lite generated code in Java to the right.

9.3.2 Synthetic (Generated)

In addition to the actual use case mentioned above, different generations of synthetically generated objects have been made to validate that the generation is correct. [Figure 9.6](#) illustrates a generated schema that contains all the possible types that our solution can represent in any object oriented language.

Synthetic.shexl	Synthetic.java
0 # Prefixes...	0 package a;
1 a:b {	1 // Imports...
2 :c xsd:string ;	2 public class B {
3 :d xsd:integer ;	3 private String c;
4 :e xsd:float ;	4 private int d;
5 :e xsd:boolean ;	5 private int e;
6 :f @a:b ;	6 private a.B f;
7 }	7 // Constructor...
	8 // Getters and Setters...
	9 }

Figure 9.6: Synthetic schema in shexl syntax to the left. And the ShEx-Lite generated code in Java to the right.

Part III

Project Synthesis

CHAPTER 10

Evaluation of Results

10.1 Methodology

In order to evaluate the proposed solutions to the two questions posed in [Chapter 1](#), the following methodologies has been used.

1. To evaluate how error detection has been improved, compare the number of actual errors found in a form by the tools detected and against the proposed solution.
2. To evaluate how the error information system has been improved, the number of elements that make up the error messages of each existing tool and of our solution is compared. In addition, a survey is carried out on different users familiar with the existing tools.
3. To evaluate to what extent we can translate shapes to domain object models we collect all the existing shapes in GitHub, reduce the set to those that fit the micro compact syntax and try to generate objects for those that are syntactically and semantically valid. In this way we can approximate what percentage we can translate.

10.2 Datasets

To test the above methodologies we will use two types of datasets. real and synthetic. In the case of the real ones, as we do not know any shape dataset, we will use the Big Query Google service to download all the files with an open source license and extension `.shex` that exist as of March 17, 2019 on GitHub. In the case of synthetic tests, schemes are designed to contain the errors described, taking into account the previous work.

10.3 Results

After evaluating the detection of errors with the methodology and the synthetic dataset, the results of [Table 10.1](#) are obtained. From the table we obtain [Figure 10.1](#) where we have eliminated row 13 to be able to scale and see the differences better.

Table 10.1: Unit results of detection of syntactic (syn.), semantic (sema.) and warning (warn.) errors produced for synthetic tests 1-13. In addition, the last row includes the aggregate sum of each column. Each test corresponds to a synthetic shape expression that contains errors or warnings. The type and number of errors that each synthetic shape contains match with the expected value from the table.

Test	Expected			ShEx-Lite			rdfshape			Shaclex			ShEx.js		
	syn.	sema.	warn.	syn.	sema.	warn.	syn.	sema.	warn.	syn.	sema.	warn.	syn.	sema.	warn.
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	2	0	0	2	0	0	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
5	0	2	0	0	2	0	0	1	0	0	1	0	0	2	0
6	0	3	0	0	3	0	0	1	0	0	1	0	0	2	0
7	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
8	2	0	0	2	0	0	1	0	0	1	0	0	1	0	0
9	0	1	1	0	1	1	0	1	0	0	1	0	0	1	0
10	1	1	1	1	0	0	1	0	0	1	0	0	1	0	0
11	0	2	1	0	2	1	0	1	1	0	1	1	0	2	1
12	1	0	1	1	0	0	1	0	0	1	0	0	1	0	0
13	0	200	200	0	200	200	0	1	0	0	1	0	0	200	0
Aggregated	5	210	207	5	209	205	4	6	1	4	6	1	4	208	1

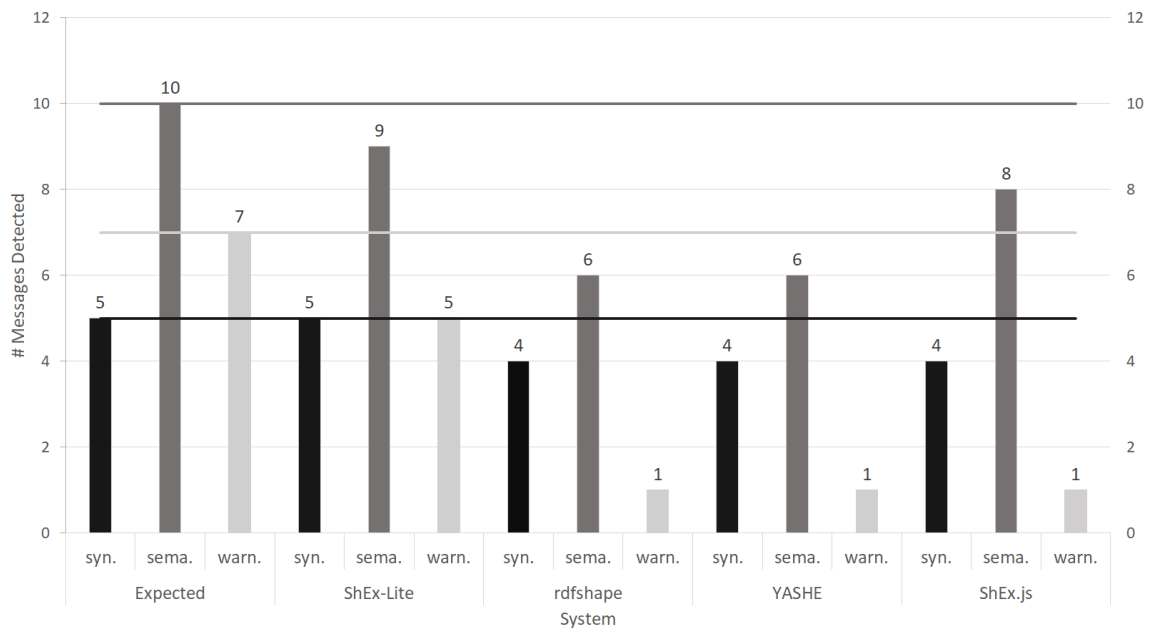


Figure 10.1: Bar chart for Table 10.1. Row 13 has been removed in order to normalize other results.

```

1 error[E007]: prefix not defined
2 --> shape_with_error_cause_pref_not_defined.shex:17:3
3   |
4 17| non_existing:label xsd:string +;
5   | ^ the prefix 'non_existing' has not been defined

```

Figure 10.2: Semantic error produced by ShEx-Lite for an undefined prefix.

```

1 Prefix 'non_existing' is not defined

```

Figure 10.3: Semantic error produced by RDFShape and YASHE for an undefined prefix.

From the results obtained and displayed, it can be seen that ShEx-Lite is, together with ShEx.js, the only system that detects multiple errors at the same time. So systems like rdfshape, based on Shaclex, or YASHE can benefit from the procedures proposed in this work.

Another important observation is that when we have a syntactical error, no system is capable of processing semantic errors or warnings that may arise. This is completely normal, since if the syntactic analysis phases are not completed, the semantic analysis cannot be performed.

Regarding the second point of our methodology, [Table 10.2](#) shows the results after evaluating the error messages (Figures 10.2 to 10.4) produced by the different systems against the good practices defined in [27].

From the results, we interpret that all the existing tools, although they really focus on data validation with Shape Expressions, would be greatly benefited from using the methodologies described in this document.

Table 10.2: Comparison of information provided in error and warning messages by ShEx-Lite, RDFShape (shaclex), YASHE and ShEx.js. The × symbol indicates that contains that feature, blank if does not contain the feature.

System	Source File	Line Number	Column Number	Code Snippet	Message Title	Message Description
ShEx-Lite	×	×	×	×	×	×
RDFShape		×	×		×	
YASHE		×	×		×	
ShEx.js					×	

```

1 error parsing input schema:
2   Parse error; unknown prefix "non_existing"

```

Figure 10.4: Semantic error produced by RDFShape for an undefined prefix.

Table 10.3: Values obtained after compiling all the elements of our real case dataset with the ShEx-Lite system. The dataset contains a total of 1.612 files of which only 19,4% contain no errors and only 2,5% contain neither errors nor warnings. Of the files without errors we were able to convert 29,4% of the documents to domain model objects. While of those who had neither errors nor warnings, we were able to translate almost 50%.

ShEx Files	1.612	100%	
Without Errors	313	19.417%	
Without Warnings	41	2.543%	
WE Transformed	92	5.707%	29.393%
WW Transformed	20	1.241%	48.780%

Regarding the translation of schemas to object models, after executing the translation on the dataset obtained from GitHub, the results found in [Table 10.3](#) are extracted.

Of these values the first that strikes us is the large number of shapes that exist on GitHub that have errors. This is explainable since we are only admitting a subset of the syntax. Therefore we can consider these 313 as those shapes that are correct expressed in ShEx micro Compact Syntax. However, of these 313 shapes that are in compact syntax, 86.9% contain some unused resource that generates warnings. On the one hand, this indicates that there is no system that warns of these things and that almost 90% of Shape Expressions users would benefit from the analysis methods developed in this work.

Furthermore, our system is capable of translating almost 30% of the schemas that do not have errors. Of the shapes that do not contain any errors or warnings, our system translates almost 50%. This tells us that the more quality a shape has, the more likely it is to be compatible with object-oriented languages.

CHAPTER 11

Planning and Budget

11.1 Planning

The planning of this work covers from the moment the proposal for the teaching commission began to be made until the moment the work is presented publicly. Of course there are some milestones that are fixed in time such as **the presentation of the proposal**, **the presentation of the dissertation** and **the defence of the work**. Therefore planning revolves around these milestones (*IDs 3, 4 and 5 from Figure 11.2*). Figure 11.1 shows a detailed timeline view of the most important events of this work.

11.1.1 Presentation of the Proposal

The acceptance of the proposal includes the first tasks (*IDs 6-8 from Figure 11.2*) in which a small investigation is done on the topics of interest and it is decided what the objectives to be pursued of the work will be.

In addition, it also includes the formal preparation of the proposal that will be delivered to the management of the computer engineering school for evaluation.

11.1.2 Presentation of the Dissertation

To consider the presentation of the dissertation as complete, it is necessary to carry out the main tasks (*IDs 9-34 from Figure 11.2*) of the work, in our case they are to carry out the

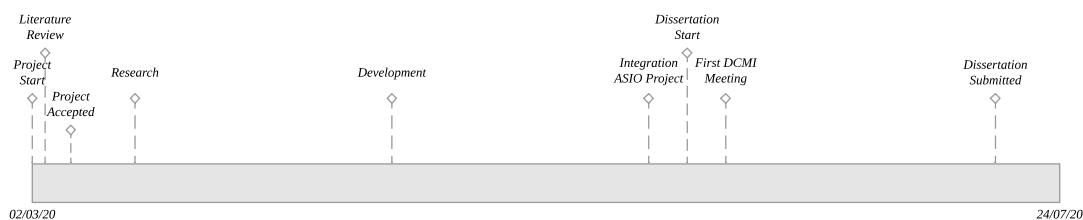


Figure 11.1: Project timeline with the most important events.

ID	ID	Outline Number	Task Mode	Task Name	Duration	Start	Finish
1		1.1		GIISOF01-4-007 GUILLERMO FACUNDO COLUNGA	105 days	Mon 02/03/20	Fri 24/07/20
2		2.1.1		Milestones	96,25 days	Fri 06/03/20	Mon 20/07/20
3	✓	3.1.1.1		Acceptance by the Teaching Commission	0 days	Fri 06/03/20	Fri 06/03/20
4	✓	4.1.1.2		Dissertation Delivery	0 days	Tue 14/07/20	Tue 14/07/20
5	✓	5.1.1.3		Viva Voice	2 hrs	Mon 20/07/20	Mon 20/07/20
6	✓	6.1.2		Proposal Preparation	3,5 days	Mon 02/03/20	Thu 05/03/20
7	✓	7.1.2.1		Aims and Objectives Research	2,5 days	Mon 02/03/20	Wed 04/03/20
8	✓	8.1.2.2		Administrative Documentation	1 day	Wed 04/03/20	Thu 05/03/20
9	✓	9.1.3		Literature Review	24 days	Mon 02/03/20	Thu 02/04/20
10	✓	10.1.3.1		RDF	5 days	Mon 02/03/20	Fri 06/03/20
11	✓	11.1.3.2		RDF Validation	2 days	Mon 09/03/20	Tue 10/03/20
12	✓	12.1.3.3		Shape Expressions	10 days	Wed 11/03/20	Tue 24/03/20
13	✓	13.1.3.4		Programming Languages	5 days	Wed 25/03/20	Tue 31/03/20
14	✓	14.1.3.5		Compilers	2 days	Wed 01/04/20	Thu 02/04/20
15	✓	15.1.4		Research and Results	45 days	Fri 03/04/20	Thu 04/06/20
16	✓	16.1.4.1		Research	20 days	Fri 03/04/20	Thu 30/04/20
17	✓	17.1.4.1.1		Error and Warning Detection	10 days	Fri 03/04/20	Thu 16/04/20
18	✓	18.1.4.1.2		Translating SHEx to ODM	10 days	Fri 17/04/20	Thu 30/04/20
19	✓	19.1.4.2		Proposed Solution Development	20 days	Fri 01/05/20	Thu 28/05/20
20	✓	20.1.4.2.1		Error and Warning Detection	15 days	Fri 01/05/20	Thu 21/05/20
21	✓	21.1.4.2.2		Translating SHEx to ODM	5 days	Fri 22/05/20	Thu 28/05/20
22	✓	22.1.4.3		Acquisition of Results	5 days	Fri 29/05/20	Thu 04/06/20
23	✓	23.1.5		Dissertation Document	25 days	Fri 05/06/20	Thu 09/07/20
24	✓	24.1.5.1		Introduction	2,5 days	Fri 05/06/20	Tue 09/06/20
25	✓	25.1.5.2		Theoretical Background	2,5 days	Tue 09/06/20	Thu 11/06/20
26	✓	26.1.5.3		Related Work	3 days	Fri 12/06/20	Tue 16/06/20
27	✓	27.1.5.4		Analysis of Existing Syntactic and Semantic Analyzers	4 days	Wed 17/06/20	Mon 22/06/20
28	✓	28.1.5.5		Proposed Syntactic and Semantic Analyzer	2 days	Tue 23/06/20	Wed 24/06/20
29	✓	29.1.5.6		Object Domain Model Translation Problem	4 days	Thu 25/06/20	Tue 30/06/20
30	✓	30.1.5.7		Proposed Translator	2 days	Wed 01/07/20	Thu 02/07/20
31	✓	31.1.5.8		Evaluation of Results	2 days	Fri 03/07/20	Mon 06/07/20
32	✓	32.1.5.9		Planning and Budget	1 day	Tue 07/07/20	Tue 07/07/20
33	✓	33.1.5.10		Conclusions	2 days	Wed 08/07/20	Thu 09/07/20
34	✓	34.1.6		Viva Voice	4 days	Fri 10/07/20	Wed 15/07/20
35	✓	35.1.6.1		Keynote Document	2 days	Fri 10/07/20	Mon 13/07/20
36	✓	36.1.6.2		Preparation of the speech	2 days	Tue 14/07/20	Wed 15/07/20

Figure 11.2: Tasks planning of the project.

corresponding research to understand the scope of the proposed objectives, to propose a solution and to obtain a few results that can be empirically testable. So that we can evaluate our solutions. And, of course, prepare the corresponding documentation that reflects all the work done.

11.1.3 Defence of the Work

The defence of the project corresponds to those tasks (*IDs 34-36 from Figure 11.2*) subsequent to the delivery of the dissertation and that have to do with public defence in which the work carried out is evaluated.

Previous sections explained each one of the different stages for this work planning. [Table 11.1](#) shows the aggregated time of each one of the major phases. You can see that most of the time is developed to the research phase that that is expected as this, even though, is a development project relies on a hard previous research.

Table 11.1: Statistics of the main project tasks.

Phase	Duration
<i>Proposal Preparation</i>	<i>3.5 days / 14 h.</i>
<i>Research</i>	<i>81.5 days / 326 h.</i>
<i>Development</i>	<i>20 days / 80 h.</i>
<i>Total</i>	<i>105 days / 420 h.</i>

11.2 Budget

To calculate this project we will take into account the estimate. From the estimation we can obtain the time that is dedicated to each of the phases, in addition we have to take into account that not all tasks are performed by the same profile and therefore not all profiles will have the same remuneration. In our case as we separated the work in three phases we will also decompose the budget in three phases. The **Proposal Preparation**, the **Research** and the **Development**.

In order to take the hourly wage we use the <https://www.salary.com> which aims to offer reliable information about hourly wages per role.

11.2.1 Proposal Preparation

The proposal preparation computes all the administrative works and the previous researches. This phase is performed by a researcher profile.

Role	Concept	Quantity	Wage / Hour	Total
<i>Researcher</i>	<i>Preparation of the proposal</i>	<i>3.5 days / 14 h.</i>	<i>55.95 E</i>	<i>783.3 E</i>
				<i>783.3 E</i>

Figure 11.3: Proposal preparation costs.

11.2.2 Research

The research phase computes research works, including the analysis performed and the writing of the dissertation. This phase is performed by a researcher profile.

Role	Concept	Quantity	Wage / Hour	Total
<i>Researcher</i>	<i>Literature Review</i>	<i>24 days / 96 h</i>	<i>55.95 E</i>	<i>783.3 E</i>
<i>Researcher</i>	<i>Research - Error and Warning Detection</i>	<i>10 days / 40 h</i>	<i>55.95 E</i>	<i>2'238.0 E</i>
<i>Researcher</i>	<i>Research - Translating ShEx to ODM</i>	<i>10 days / 40 h</i>	<i>55.95 E</i>	<i>2'238.0 E</i>
<i>Researcher</i>	<i>Documentation</i>	<i>29 days / 116 h</i>	<i>55.95 E</i>	<i>6'490.2 E</i>
				<i>11'749.5 E</i>

Figure 11.4: Research costs.

11.2.3 Development

The development tasks are done to create a Probe Of Concept (POC) that validates the proposed solution. This task are not carried out by a researcher but by an Scala Software Developer.

Role	Concept	Quantity	Wage / Hour	Total
<i>Developer</i>	<i>Error and Warning Detection System</i>	<i>15 days / 60 h</i>	<i>42.8 E</i>	<i>2'568.0 E</i>
<i>Developer</i>	<i>Translator ShEx to ODM System</i>	<i>5 days / 20 h</i>	<i>42.8 E</i>	<i>856.0 E</i>
				<i>3'424 E</i>

Figure 11.5: Development costs.

11.2.4 Aggregated Costs

After calculating the partial costs of each of the phases of the project, the costs are added, obtaining the value of the real cost of executing the project. To this is added a 10% from adding all the indirect costs of the project and the corresponding taxes. With all this we obtain the final cost of our project. It is important to remember that being a research project, the benefits are the project itself and this is a cost estimate.

Phase		Total
<i>Proposal Preparation</i>		<i>783.3 E</i>
<i>Research</i>		<i>11'749.5 E</i>
<i>Development</i>		<i>3'424.0 E</i>
<i>Project Cost</i>		<i>15'956.8 E</i>
<i>Project Indirect Cost</i>	<i>10 %</i>	<i>1'595.68 E</i>
<i>Taxes</i>	<i>21 %</i>	<i>3'350.93 E</i>
<i>Project Total Cost</i>		<i>20'903.41 E</i>

Figure 11.6: Aggregated costs.

CHAPTER 12

Conclusions

The proposed analysis methods based on static syntactic and semantic analysis techniques can be used to build different tools for software development, such as development environments. Furthermore, after evaluating the results, we observe that up to 90% of ShEx users can benefit from the techniques described. And that tools like RDFShape, YASHE or ShEx.js can adopt the techniques described here to improve their static code analysis capabilities.

The analysis and proposed methods for transforming schemas into models based on object-oriented technologies can be used to link schemas made through Shape Expressions with business applications. An example of this is the European project ASIO Hercules. Where the system proposed in this document is used to automatically join an ontology with Java-based applications. The results obtained in the ASIO Hercules European project have resulted in the paper "*ASIO: a Research Management System based on Semantic technologies*" sent to the CIKM¹ 2020 conference. In it, among other things, it explains the key role of the system proposed in this dissertation to unite an Ontological Infrastructure with a Semantic Architecture.

The two solutions, the analyser and the translator, are embodied in a single system called ShEx-Lite that includes the implementation of lexical, syntactic and semantic analysers. And as code generation, translation into object-oriented languages. This system has given as one of its results the article "ShEx-Lite: Automatic generation of domain object models from Shape Expressions" that has been sent to the ISWC20² conference.

12.1 Future Work

This work opens new future lines of research that we plan to work on. What follows is a brief description of such works.

¹<https://www.cikm2020.org/>

²<https://iswc2020.semanticweb.org/>

12.1.1 Implementation of New Analyses

Currently we do not have an existing documentation on the most common errors in ShEx and therefore it is difficult to generate new analyses, it is not known what to look for. However, with the methods exploited in this project, you can start, for example, to explore all open source schemes. From them obtain data that serves to develop documentation on the most common errors or new analyses.

12.1.2 Automatic Error Fixing

The described system detects errors and locates them at a specific point in the representation, through pattern recognition. The next step, once we have automatically detected the errors, would be to develop a system that automatically corrects these errors by means of transformations on the representations.

This opens the door to new lines of research, since the representation of the schemas must be modified without affecting its schemas semantics.

12.1.3 Machine Learning to Recognize Patterns

As explained in [33], systems based on machine learning can be trained with our intermediate schema representations, and then automatically classify the schemas according to the training criteria.

12.1.4 New Input Syntaxes for Translation to Aspect Oriented Languages

As a line of future research but it is already underway. The Dublin Core research group is working on defining a ShEx Micro based syntax with tabular representation, mainly spreadsheets. This would allow to attract a much larger user base. As a proposal of the group itself is to include this syntax as input to our ShEx-Lite system in order to enhance all analytics and generation. For this we hold telematic meetings every other Wednesday.

12.1.5 New Target Languages

ShEx-Lite only generates Java and Python code. However, adding the ability to generate in new programming languages would increase the ability to integrate with more systems.

Part IV

Annexes and References

ShEx Micro Language

A.1 Syntax Specification

```

1 Schema { start:shapeExpr? shapes:[shapeExpr+]? }
2 shapeExpr = NodeConstraint | Shape ;
3 shapeExprLabel = IRIREF ;
4 NodeConstraint { id:shapeExprLabel nodeKind:("iri" | "bnode" | "nonliteral"
5 | "literal")? datatype:IRIREF? numericFacet*
6 values:[valueSetValue+]? }
7 numericFacet = (mininclusive|minexclusive|maxinclusive|maxexclusive):
8 numericLiteral
9 numericLiteral = INTEGER | DECIMAL | DOUBLE ;
10 valueSetValue = objectValue | IriStem ;
11 objectValue = IRIREF | ObjectLiteral ;
12 ObjectLiteral { value:STRING language:STRING? type:STRING? }
13 IriStem { stem:IRIREF }
14 Shape { id:shapeExprLabel expression:tripleExpr}
15 tripleExpr = EachOf | TripleConstraint ;
16 EachOf { expressions:[tripleExpr{2,}] }
17 TripleConstraint { predicate:IRIREF valueExpr:shapeExpr? min:INTEGER?
18 max:INTEGER}

```

A.2 Lexical Specification

```

1 IRIREF : (PN_CHARS | '._ | ',' | ',/' | '\\' | '#' | '@' | '%' | '&' | UCHAR)* ;
2 BNODE : '._' (PN_CHARS_U | [0-9]) ((PN_CHARS | '._)* PN_CHARS)? ;
3 BOOL : "true" | "false" ;
4 INTEGER : [+]? [0-9] + ;
5 DECIMAL : [+]? [0-9]* '._' [0-9] + ;
6 DOUBLE : [+]? ([0-9] + '._' [0-9]* EXPONENT | '._' [0-9]+ EXPONENT | [0-9]+
7 EXPONENT) ;
8 LANGTAG : ([a-zA-Z])+('-'([a-zA-Z0-9]))* ;
9 STRING : .* ;
10
11 PN_PREFIX : PN_CHARS_BASE ((PN_CHARS | '._)* PN_CHARS)? ;
12 PN_CHARS_BASE : [A-Z] | [a-z] | [\u00C0-\u00D6] | [\u00D8-\u00F6]
13 | [\u00F8-\u02FF] | [\u0370-\u037D] | [\u037F-\u1FFF]
14 | [\u200C-\u200D] | [\u2070-\u218F] | [\u2C00-\u2FEF]
15 | [\u3001-\uD7FF] | [\uF900-\uFDCF] | [\uFDF0-\uFFFD]
16 | [\u10000-\uEFFFE] ;
17 PN_CHARS : PN_CHARS_U | '._' | [0-9] | '\u00B7' | [\u0300-\u036F] |

```

```
18 [\u203F-\u2040] ;
19 PN_CHARS_U      :      PN_CHARS_BASE | ' _ ' ;
20 UCHAR          :      '\\u' HEX HEX HEX HEX
21 | '\\U' HEX HEX HEX HEX HEX HEX HEX HEX ;
22 HEX            :      [0-9] | [A-F] | [a-f] ;
23 EXPONENT       :      [eE] [+]? [0-9]+ ;
```

APPENDIX B

ShEx-Lite Antlr Grammar

B.1 Syntax Specification

```
1 // KEYWORDS
2
3 //A:           'a';
4 ANYTYPE:      '.';
5 BASE:         'base';
6 BNODE:        'bnode';
7 IRI:          'iri';
8 LITERAL:      'literal';
9 NONLITERAL:   'nonliteral';
10 PREFIX:      'prefix';
11 START:       'start';
12 IMPORT:      'import';
13
14 // Literals
15 // Meant to be extended with interpolated text. (from Swift 3)
16 STRING_LITERAL:    STATIC_STRING_LITERAL;
17 STATIC_STRING_LITERAL:  '"' Quoted_text? '"';
18 IRI_LITERAL:       '<' (~[\u0000-\u0020=<>"{}|~'\`] | Unsigned_character)* '>';
19 DECIMAL_LITERAL:  ('0' | [1-9] (Digits? | '_' + Digits)) [1L]?;
20 FLOAT_LITERAL
21 :                (Digits '.' Digits? | '.' Digits) Exponent_part? [fFdD]?
22 |                Digits (Exponent_part [fFdD]? | [fFdD])
23 ;
24
25 // Separators
26 LPAREN:          '(';
27 RPAREN:          ')';
28 LBRACE:          '{';
29 RBRACE:          '}';
30 LBRACK:          '[';
31 RBRACK:          ']';
32 SEMI:           ';';
33 COLON :         ':';
34 COMMA:          ',';
35
36 // Operators
37 AT:              '@';
38 ADD:            '+';
39 EQ:             '=';
40 MUL:            '*';
```

```

41 QUESTION:          '??';
42
43 // Comments and Whitespace
44 COMMENT:           ('#' ~[\r\n]* | '/'* (~[*] | '*' ('\\/' | ~[/]))* '*/') -> channel(HIDDEN);
45 WHITE_SPACE:       [ \t\r\n\f]+ -> channel(HIDDEN);
46
47 // Identifiers
48 IDENTIFIER:        Identifier_head Identifier_characters?;
49
50 fragment Identifier_head
51 : [a-zA-Z]
52 | '_'
53 | '\u00A8' | '\u00AA' | '\u00AD' | '\u00AF' | [\u00B2-\u00B5] | [\u00B7-\u00BA]
54 | [\u00BC-\u00BE] | [\u00C0-\u00D6] | [\u00D8-\u00F6] | [\u00F8-\u00FF]
55 | [\u0100-\u02FF] | [\u0370-\u167F] | [\u1681-\u180D] | [\u180F-\u1DBF]
56 | [\u1E00-\u1FFF]
57 | [\u200B-\u200D] | [\u202A-\u202E] | [\u203F-\u2040] | '\u2054' | [\u2060-\u206F]
58 | [\u2070-\u20CF] | [\u2100-\u218F] | [\u2460-\u24FF] | [\u2776-\u2793]
59 | [\u2C00-\u2DFF] | [\u2E80-\u2FFF]
60 | [\u3004-\u3007] | [\u3021-\u302F] | [\u3031-\u303F] | [\u3040-\uD7FF]
61 | [\uF900-\uFD3D] | [\uFD40-\uFDCF] | [\uFDF0-\uFE1F] | [\uFE30-\uFE44]
62 | [\uFE47-\uFFFD]
63 ;
64
65 fragment Identifier_characters
66 : Identifier_character+
67 ;
68
69 fragment Identifier_character
70 : [0-9]
71 | [\u0300-\u036F]
72 | [\u1DC0-\u1DFF]
73 | [\u20D0-\u20FF]
74 | [\uFE20-\uFE2F]
75 | Identifier_head
76 ;
77
78 // Fragment rules
79
80 fragment Quoted_text
81 : Quoted_text_item+
82 ;
83
84 fragment Quoted_text_item
85 : Escaped_character
86 | ~["\n\r\\]
87 ;
88
89
90 fragment Escaped_character
91 : '\\\'' [0\t\nr"']
92 | '\\x' Hexadecimal_digit Hexadecimal_digit
93 | '\\u' '{' Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit '}',
94 | '\\u' '{' Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit
95   Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit '}',
96 ;
97

```

```
98 fragment Unsigned_character
99 : '\\u' Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit
100 | '\\u' Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit
101   Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit Hexadecimal_digit
102 ;
103
104 fragment Digits
105 : Digit ([0-9_]* Digit)?
106 ;
107
108 fragment Digit
109 : [0-9]
110 ;
111
112 fragment Exponent_part
113 : [eE] [+]? Digits
114 ;
115
116 fragment Hexadecimal_digits
117 : Hexadecimal_digit ((Hexadecimal_digit | '_')* Hexadecimal_digit)?
118 ;
119
120 fragment Hexadecimal_digit
121 : [0-9a-fA-F]
122 ;
```

B.2 Lexical Specification

```
1 schema
2 : statement+ EOF
3 ;
4
5 statement
6 : import_stmt
7 | definition_stmt
8 ;
9
10 import_stmt
11 : IMPORT iri=literal_iri_value_expr
12 ;
13
14 definition_stmt
15 : start_def_stmt
16 | prefix_def_stmt
17 | base_def_stmt
18 | shape_def_stmt
19 ;
20
21 start_def_stmt
22 : START EQ shape=call_shape_expr
23 ;
24
25 prefix_def_stmt
26 : PREFIX IDENTIFIER? COLON iri=literal_iri_value_expr
27 ;
```

```
28
29 base_def_stmt
30   : BASE iri=literal_iri_value_expr
31   ;
32
33 shape_def_stmt
34   : label=call_prefix_expr expr=constraint_expr
35   ;
36
37 expression
38   : literal_expr
39   | cardinality_expr
40   | constraint_expr
41   ;
42
43 literal_expr
44   : literal_real_value_expr
45   | literal_string_value_expr
46   | literal_iri_value_expr
47   ;
48
49 literal_real_value_expr
50   : DECIMAL_LITERAL
51   ;
52
53 literal_string_value_expr
54   : STRING_LITERAL
55   ;
56
57 literal_iri_value_expr
58   : IRI_LITERAL
59   ;
60
61 cardinality_expr
62   : MUL
63   | ADD
64   | QUESTION
65   | LBRACE min=DECIMAL_LITERAL RBRACE
66   | LBRACE min=DECIMAL_LITERAL COMMA max=DECIMAL_LITERAL RBRACE
67   | LBRACE min=DECIMAL_LITERAL COMMA RBRACE
68   ;
69
70 constraint_expr
71   : constraint_node_expr
72   | constraint_block_triple_expr
73   | constraint_triple_expr
74   ;
75
76 constraint_node_expr
77   : constraint_node_iri_expr
78   | constraint_valid_value_set_type
79   | constraint_node_any_type_expr
80   | call_expr
81   | constraint_node_non_literal_expr
82   | constraint_value_set_expr
83   | constraint_node_bnode_expr
84   | constraint_node_literal_expr
```

```
85 ;
86
87 constraint_block_triple_expr
88 : LBRACE (constraint_triple_expr)+ RBRACE
89 ;
90
91 constraint_triple_expr
92 : property=call_prefix_expr constraint=constraint_node_expr cardinality=cardinality_expr?
93     SEMI?
94 ;
95
96 constraint_node_iri_expr
97 : IRI
98 ;
99
100 constraint_valid_value_set_type
101 : call_prefix_expr
102 | call_shape_expr
103 | literal_string_value_expr
104 | literal_real_value_expr
105 ;
106
107 constraint_node_any_type_expr
108 : ANYTYPE
109 ;
110
111 constraint_node_non_literal_expr
112 : NONLITERAL
113 ;
114
115 constraint_value_set_expr
116 : LBRACK constraint_valid_value_set_type* RBRACK
117 ;
118
119 constraint_node_bnode_expr
120 : BNODE
121 ;
122
123 constraint_node_literal_expr
124 : LITERAL
125 ;
126
127 call_expr
128 : call_prefix_expr
129 | call_shape_expr
130 ;
131
132 call_prefix_expr
133 : pref_lbl=IDENTIFIER? COLON shape_lbl=IDENTIFIER
134 | base_relative_lbl=literal_iri_value_expr
135 ;
136
137 call_shape_expr
138 : AT prefix_lbl=IDENTIFIER? COLON shape_lbl=IDENTIFIER
139 | AT base_relative_lbl=literal_iri_value_expr
140 ;
```

GitHub CI Workflow

```
1 name: ShEx-Lite CI
2
3 on: [push, pull_request]
4
5
6 jobs:
7
8   # Test run on linux platform
9   ubuntu-test:
10
11     # Job name
12     name: Test Linux Platform
13
14     # This jobs runs on linux
15     runs-on: ubuntu-latest
16
17     steps:
18       - uses: actions/checkout@v2
19       - name: Set up JDK 12
20         uses: actions/setup-java@v1
21         with:
22           java-version: 12
23           architecture: x64
24       - name: Run tests
25         run: sbt ++2.13.1! clean test
26
27   # Test run on linux platform
28   windows-test:
29
30     # Job name
31     name: Test Windows Platform
32
33     # This jobs runs on linux
34     runs-on: windows-latest
35
36     steps:
37       - uses: actions/checkout@v2
38       - name: Set up JDK 12
39         uses: actions/setup-java@v1
40         with:
41           java-version: 12
42           architecture: x64
43       - name: Run tests
```



```
44         run: sbt ++2.13.1! clean test
45
46     # Test run on linux platform
47     macos-test:
48
49     # Job name
50     name: Test OS X Platform
51
52     # This jobs runs on linux
53     runs-on: macos-latest
54
55     steps:
56     - uses: actions/checkout@v2
57     - name: Set up JDK 12
58       uses: actions/setup-java@v1
59       with:
60         java-version: 12
61         architecture: x64
62     - name: Install sbt
63       run: brew install sbt
64     - name: Run tests
65       run: sbt ++2.13.1! clean test
66
67     # Test run on linux platform
68     coverage-report:
69
70     # Job name
71     name: Coverage Report
72
73     # This jobs runs on linux
74     runs-on: ubuntu-latest
75
76     steps:
77     - uses: actions/checkout@v2
78     - name: Set up JDK 12
79       uses: actions/setup-java@v1
80       with:
81         java-version: 12
82         architecture: x64
83     - name: Run tests with coverage
84       run: sbt ++2.12.10! clean coverage test coverageReport
85     - name: Upload coverage report to codecov
86       uses: codecov/codecov-action@v1
87       with:
88         fail_ci_if_error: true
```

Bibliography

- [1] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile networks and applications*, 19(2):171–209, 2014.
- [2] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*, pages 42–47. IEEE, 2013.
- [3] Mark Levene and George Loizou. A graph-based data model and its ramifications. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):809–823, 1995.
- [4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.
- [5] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.
- [6] Jose Emilio Labra Gayo, Eric Prud’Hommeaux, Iovka Boneva, and Dimitris Kontokostas. Validating rdf data. *Synthesis Lectures on Semantic Web: Theory and Technology*, 7(1):1–328, 2017.
- [7] Arthur G Ryman, Arnaud Le Hors, and Steve Speicher. Oslc resource shape: A language for defining constraints on linked data. *LDOW*, 996, 2013.
- [8] Holger Knublauch. Spin-modeling vocabulary. *W3C Member Submission*, 22, 2011.
- [9] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, 1(1):7–87, 1990.
- [10] List of object-oriented programming languages, April 2020. Page Version ID: 950660258.
- [11] Eric Prud’hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. Shape expressions: an rdf validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 32–40, 2014.
- [12] Programming language, May 2020. Page Version ID: 958722580.
- [13] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.
- [14] Tim Berners-Lee et al. Semantic web road map, 1998.
- [15] Eric Van der Vlist. *Relax ng: A simpler schema language for xml*. " O’Reilly Media, Inc.", 2003.

-
- [16] Domain-specific language, March 2020. Page Version ID: 945660040.
- [17] Turing completeness, May 2020. Page Version ID: 954916159.
- [18] Lexical analysis, May 2020. Page Version ID: 955735363.
- [19] Optimizing compiler, April 2020. Page Version ID: 949660274.
- [20] Entorno de desarrollo integrado, May 2020. Page Version ID: 126109294.
- [21] dotnet/roslyn. Library Catalog: github.com.
- [22] Jose Emilio Labra-Gayo, Herminio García-González, Daniel Fernández-Alvarez, and Eric Prud'hommeaux. Challenges in rdf validation. In *Current Trends in Semantic Web Technologies: Theory and Practice*, pages 121–151. Springer, 2019.
- [23] Iovka Boneva, Jérémie Dusart, Daniel Fernández Alvarez, and Jose Emilio Labra Gayo. Semi automatic construction of shex and shacl schemas. 2019.
- [24] Daniel Fernández-Alvarez, Jose Emilio Labra-Gayo, and Herminio Garcia-González. Inference and serialization of latent graph schemata using shex, 2016.
- [25] Herminio Garcia-Gonzalez, Daniel Fernandez-Alvarez, and Jose Emilio. Shexml: An heterogeneous data mapping language based on shex.
- [26] Robert W Floyd. Syntactic analysis and operator precedence. *Journal of the ACM (JACM)*, 10(3):316–333, 1963.
- [27] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.
- [28] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
- [29] Martin Fowler. *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.
- [30] Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.
- [31] Veljko Milutinovic and Milos Kotlar. *Exploring the DataFlow Supercomputing Paradigm: Example Algorithms for Selected Applications*. Springer, 2019.
- [32] Xsd simple elements.
- [33] Oscar Rodríguez Prieto. *BigCode Intrastructure for Building Tools to Improve Software Development*. PhD thesis, University of Oviedo, 2020.