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

GUILLERMO FACUNDO COLUNGA

A DISSERTATION SUBMITTED FOR THE DEGREE OF SOFTWARE ENGINEER

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF OVIEDO

2020

Acknowledgments

Write your acknowledgments here.

Abstract

Surface integration is an important step for automatic 3D reconstruction of real objects. The goal of a surface integration algorithm is to reconstruct a surface from a set of range images registered in a common coordinate system. Based on the surface representation used, existing algorithms can be divided into two categories: volume-based and mesh-based. Volume-based methods have been shown to be robust to scanner noise and small features (regions of high curvature) and can build water tight models of high quality. It is, however, difficult to choose the appropriate voxel size when the input range images have both small features and large registration errors compared to the sampling density of range images. Mesh-based methods are more efficient and need less memory compared to volume-based methods but these methods fail in the presence of small features and are not robust to scanning noise.

This paper presents a robust algorithm for mesh-based surface integration of a set of range images. The algorithm is incremental and operates on a range image and the model reconstructed so far. Our algorithm first, transform the model in the coordinate system of the range image. Then, it finds the regions of model overlapping with the range image. This is done by shooting rays from the scanner, through the vertices in the range image and intersecting them with the model. Finally, the algorithm integrates the overlapping regions by using weighted average of points in the model and the range image. The weights are computed using the scanner uncertainty and helps in reducing the effects of scanning noise. To handle small features robustly the integration of overlapping regions is done by computing the position of vertices in the range image along the scanner's line of sight. Since for every point in a range image there is exactly one depth value, the reconstructed surface in the regions of high curvature will not have self-intersections.

Contents

List of Figures

List of Tables

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 forms 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 application 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.

In order to solve this key concept 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 URIs[1] offered a solution to the data integration problem as RDF datasets produced by different means can seamlessly be integrated with other data.

Also, and 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:

- Use URIs as names for things.

---

[1] A 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
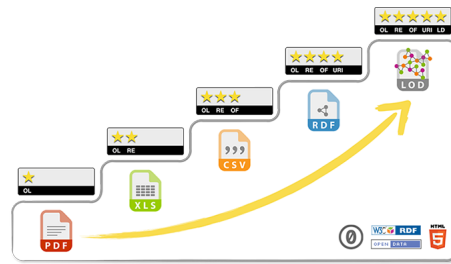
Figure 1.1: The 5 star steps of Linked Data.

- 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.

This four principles are called 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 dilema**.

RDF production/consumption dilema 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 dilema 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, SPIN, 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 genome *(a real use case of ShEx)*. A language with such a number of possibilities requieres from a strong sintactic and semantic validation and that leads us to our first research question.

Research Question 1. *How much the existing sintactic and semantic validation systems for shape expressions can be enhanced?*

Secondly and very related to programing languages, if we take the PopularitY of Programming Language *(PYPL)* Index[2] from June 2020 we can see that more than half of the share is occupied by languages that support the object orienten paradigm. And therefore this paradigm becomes the most used one. The aim of this paradigm is to model real world domains, acocrding to [8]. 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 research question relies on this and tries to automatically transform shape expressions into object domain models coded in any language that supports the object oriented paradigm:

Research Question 2. *Till which point can we automatically translate existing shape expressions in 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 [3]. This version is a strict subset of the complete ShEx grammar and therefore any conclusion 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.

---

[2]http://pypl.github.io/PYPL.html

[3]https://dcmi.github.io/dcap/shex_lite/micro-spec.html

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 analyzer 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 analyzer that performs both sintactic and semantic analysis so it produces human-friendly errors and warnings that users can use to fix their schemas.

3. Automatic translation of schemas in to object domain models in `Java` and `Python`. The proposed system integrates an open back-end with build-in code translation from the validated schemas to domain models in object oriented programming languages *(OOPL)* [9].

4. Evaluation of errors and warning generated of our proposed soulution 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 7 Contains a detailed initial planning and budget for the project, this is the designed planning followed during the execution of the project and the initial estimated budget.

Chapter 8 Gives a basic theoretical background that it is needed to fully understand the concepts explained in the following chapters.

Chapter 9 Provides a technical description of the design and implementation of the compiler itself. This includes, analysis, design, the technological stack choices, diagrams, implementation decisions and tests.

Chapter 10 Compares the initial planning developed in chapter 4 with the final one. This includes the genuine execution planning of the project and the reasons and events that modified the one from chapter 4.

CHAPTER 2

Teoretical 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 fundaments of Linked Data, RDF, RDF Validation, programing languages and compilers. This sections is devoted to carefully explain those concepts to the needed deepth to fully understand this dissertation, but for those readers that want a deeper explanation a more detailed view of the concepts presented here is offered in [6, 10, 11].

2.1 RDF

Resource Description Framework (RDF) is a standard model for data interchange on the web, started in 1998 and the first version of the specification was published in 2004 by the W3C according to [12]. 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 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"). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications. 2.3 shows an example of how different triples can be use to compose a graph, this graph represents the same as the Figure 2.2

This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest

---

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 exmaple shows the n-triples that generate the graph from Figure 2.3.
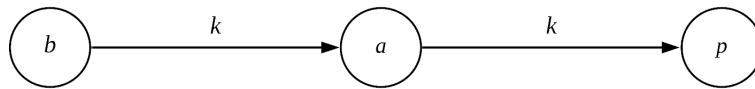
Figure 2.3: RDF graph formed by triplets from Figure 2.2, where b corresponds to *<http://example/bob>*, a corresponds to *<http://example/alice>*, p corresponds to *<http://example/peter>* and k corresponds to *<http://example/knows>*.

possible mental model for RDF and is often used in easy-to-understand visual explanations.

Also, related to this we strongly recommend the Tim Berners-Lee's writings on Web Design Issues [13] where he explain the issues of the liked 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 **??**, this dissertation will focus on Shape Expressions. But in order to validate RDF data every technology will need 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; and
- the possible values associated with those arcs.
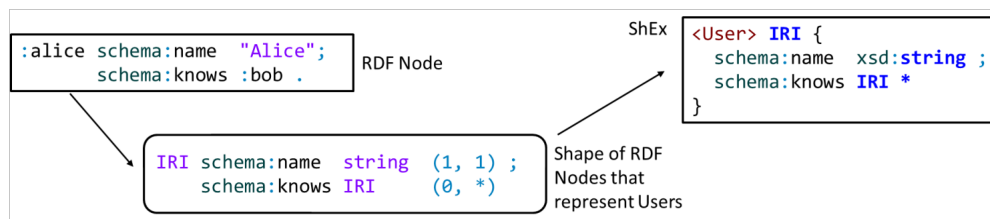
Figure 2.4: RDF node and its shape.

```
 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 describes a user as a node that has one name of type string, an optional bithd date of type date, one gende of type Male, Female or free string and a set between 0 and infinite of other users represented by the knows property.

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.

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 [14]. The language was described in a paper [10] 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 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.

- `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.
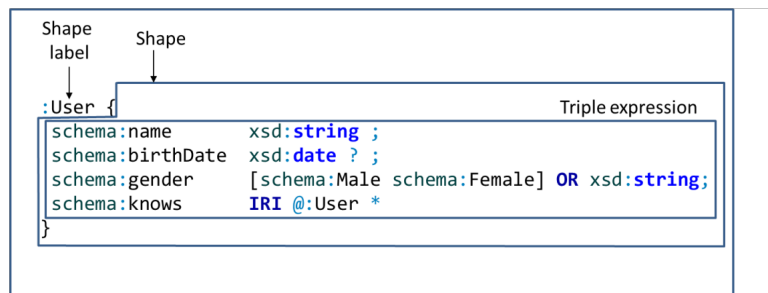
Figure 2.6: Shapes, shape expression labels and triple expressions.
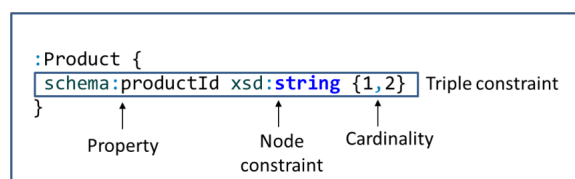
Figure 2.7: Parts of a triple expression.

- 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 labeled 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 asing to the `:User` label. Then, the shape is composed by triple expressions. The triple expression structure is explained in **??**, 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 containts 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 is the notion of conformance with schema. Even one is using ShEx to create forms, 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 FHIR, 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., doing 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   Check links.

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.

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 we will just explore the existence of them as it is very interesting to know how other tools approach the same issue.

SHACL

Also in [6], Chapter 5, it is fully explained that 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 are 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 [11] "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, General Purpose Languages (GPL) and Domain Specific Languages (DSL). The main difference overtime is that, as said in [15], 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 deep we would classified it as a Declarative one, that means that it is not Touring Complete [16].

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).
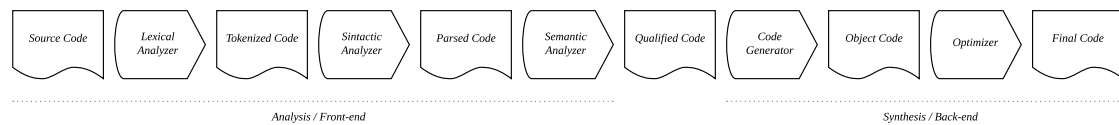
Figure 2.8: Compiler stages.

2.4.1   Internal Structure

In order to decompose the internal structure of a compiler they have been split in to the most common task they do Figure 2.8, of course this doesn't mean that there are compilers with more or less stages, but at the end everything can be group into any of the groups that we will explain:

Lexycal Analyzer

The lexical analyzer task is to get the input and split it in to tokens [17], which are build from lexemes. If the compiler cannot find a valid token for some lexemes in the source code will generate an error, as the input cannot be recognized.

Syntactic Analyzer

The syntactic analyzer takes the tokens generated during the lexical analysis and parses them in such a way that try's to group 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 Analyzer

The semantic analyzer has two main tasks, usually. First it validates that the source code semantics are correct, for example 4 + "aaa" would not make sense. 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 [18] 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 compiler 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 have been the "old-fashion" but it presents some drawbacks:

- A poor IDE [19] 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 where 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 lead 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 where a blackbox 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 [20]. This concept has been perfectly accepted and solved many problems. 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. This means that 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 overthe main studies related to our project, exploring what they have achieved and someof their limitations.

3.1 Simplifications of ShEx

3.1.1 The S language

In 2019 at [21] 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 [22]. 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 Doublin Core Team[1] 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 contratints. This specification is not official and has not been validated yet but it is very importat for our work as we will also work in a simplification of a syntax of ShEx.

And to the best of our knowledge and after the research process carried out for this project no

---

[1]https://dublincore.org/

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 thinks 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.

This kind of tools can benefit from SHEx-Lite from the point of view that new functionalities can be easily implemented and tested in the lite version of the language before even touching the stable releases of both tools. In the case of ShEx this is more obvious as ShEx-Lite and ShEx are both implemented in Scala and if good design principles are used functionalities can be just migrated and expanded for the rest of the language.

The most important validators are:

Shaclex

According to the Shaclex[2] 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 Dr. 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 completly inspired by it and aims to transfer the sintactic an semantic validation enhancements to it.

ShEx.js

Another example of os 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.

---

[2]https://github.com/weso/shaclex

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.

This tools will completely benefit from ShEx-Lite and there are currently collaborations in process. 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 coloring syntax or incremental compilation are available.

YASHE

YASHE[3] (Yet Another ShEx Editor), is a Shape Expressions IDE which started as a fork ofYASQE(which is based on SPARQL). This tool performs lexical and syntactic analysis of the content of the editor, thus offering the user a realtime syntactic error detector. It has features like: 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 dition on their own software so they became another ShEx IDE.

VSCode

VSCode is a source code light-weight editor developed by Micorsoft 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.

---

[3]https://github.com/weso/YASHE
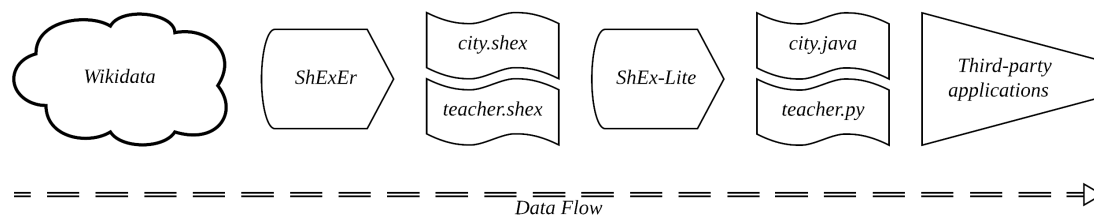
Figure 3.1: ShEx-Lite integration with Shexer for automatically generating java domain object models for the Wikidata schemaless existing data. This shoes the schemaless data from wikidata from which shape expressions are infeered by shexer and later transformed to java plain objects by means of ShEx-Lite so third party apllications can implement the domain model.

Shexer

Shexer[4] is a python library aimed to perform automatic automatic extraction of schemas in both ShEx and SHACL from an RDF input graph. That is if all the other tools take the schemas as the input and validate a graph with it, this tool takes a graph and from it it infers the schemas that it might contain. Its work is fully described in [22, 23].

ShExML

ShExML[5] 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 [24].

An example of how this different tools can work together thanks to ShEx-Lite would be the following, illustrated at Figure 3.1. Wikidata currently holds millions of registers that do not have any schema that validates them. And they need to make consumer that represents the data in to an object domain model. Without any tool this is just almost impossible, but this 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 favorite OOL.

---

[4]https://github.com/DaniFdezAlvarez/shexer

[5]https://github.com/herminiogg/ShExML

Part I

Enhancing Error and Warning Detection and Emition on ShEx

CHAPTER 4

Analysis of Existing Sintactic and Semantic Analizers

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 emition. 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 favors the security of the language will be chosen. For example. If the specification did not say 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 sintactic restrictions 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.

```
1 prefix schema : <http://schema.org/>
2 prefixx : <http://schemata.org/>
```

(a) Sintactic error at line 2 because the prefix keyword contains a typo. (Keyword Sintax Error)

```
1 prefix schema : <http://schema.org/>
2 prefix : http://schemata.org/
```

(b) Sintactic error at line 2 because the IRI does not follow the IRI pattern. (Literal Syntax Error)

```
1 prefix : <http://example/>
2 prefix xsd: <http://schema.org/>
3
4 :Picture
5  :name  xsd:string ;
6  :width  xsd:integer
7 }
```

(c) Sintactic error at line 3 because the Shape definition requires an open bracket. (Struct Syntax Error)

```
1 prefix xsd: <http://example/>
2 prefix xsd: <http://schama.org/>
```

(d) Semantic error at line 2 because the prefix override is not allowed.

```
1 prefix : <http://example/>
2 prefix xsd: <http://schama.org/>
3
4 :Picture {
5  :name  xsd:string ;
6  cdt:width  xsd:integer
7 }
```

(e) Semantic error at line 6 as the prefix cdt is not defined.

```
1 prefix : <http://example/>
2 prefix xsd: <http://schama.org/>
3
4 :User {
5  :name  xsd:string ;
6  :knows  @:Person *
7 }
```

(f) Semantic error at line 6 as the shape Person is not defined.

```
1 prefix xsd: <http://example/>
2 prefix    : <http://schama.org/>
3
4 :User {
5  :name :string
6 }
```

(g) Semantic warning at line 1 because the prefix is never used.

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*. To others within the previous elements you will 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 this errors.

4.2   Sintactic Analyzers

According to [25] we consider a Sintactic Analyzer a piece of software capable of parse, generate a parse tree and detect and emmit sintactic 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 analyzed 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 keywork `prefix` by adding an extra `x` the error obtained is: `This line is invalid.  Expected: PNAME_NS.`

To our point of view this error message nor is not correct because it does not provide the

Table 4.1: Detection of the different sintactic errors by the current existing ShEx tools that sintactically analyze the shape expressions.

	Sintactic Errors							
Analyzers	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 analyze the shape expressions.

	Semantic Errors						
Analyzers	Prefix Override	Base Override	Start Shape Override	Shape Override	Non Existing Prefix Reference	Non Existing Base Reference	Non Existing 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]

user enough information to fix the schema.

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

4.3 Semantic Analyzers

As Semantic Analyzers 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 analyzed 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 drawbaks. 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 previus 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 [26].** Existing error messages, originated both by sintactic 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 analyze if declared resources are used and therefore there is no need to generate warnings. We propose to not only fully analyze the resources to detect non-used ones but also the creation or 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 overriden, 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 enchace 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 analyzers 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 Sintactic and Semantic Analyzer

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

5.1 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 5.1 shows the different components of this analyzer.

5.1.1 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
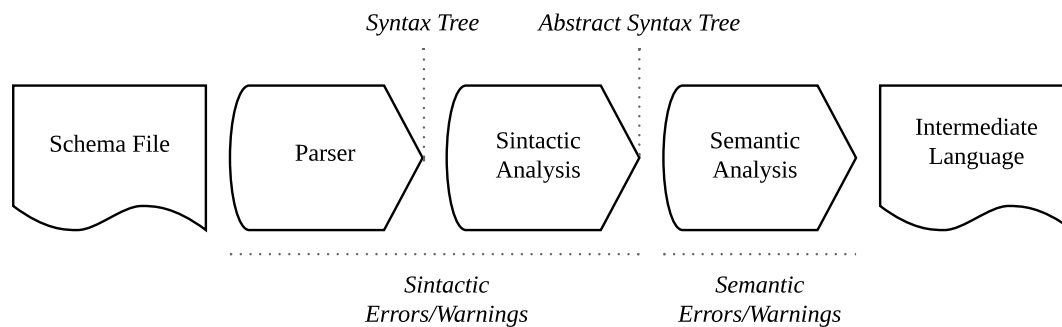
Figure 5.1: Sintactic and Semantic Analyzer structure.
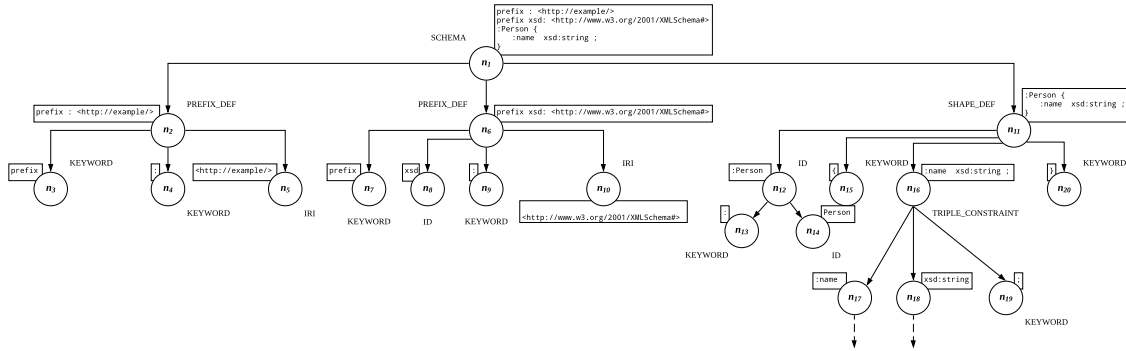
Figure 5.2: Syntax Tree tweenty first nodes produced by the parser.

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 5.2 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 5.2 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.

5.1.2 Sintactic Analyzer

The sintactic analyzer 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 tranform the Syntax Tree Figure 5.2 into an Abstract Syntax Tree Figure 5.3 *(without the green and red relations)*.

For this, each node within our syntactic tree is aware of the context in which it is. Therefore we can ask questions to the nodes, such as to a prefix definition *(Figure 5.2 n_2)*, do you have a label? *(No)* or who is the node that defines your iri? *(node n_5)*. With questions like these, the syntactic tree can be analyzed for patterns that represent warnings or errors.

5.1.3 Semantic Analyzer

The semantic analyzer is responsible of building all the possible relations between the AST nodes, analyze and check that all those relations that must exist indeed exist. For this porpouse as just seen we reduce our Syntax Tree to an Abstract Syntax Tree. Figure 5.3
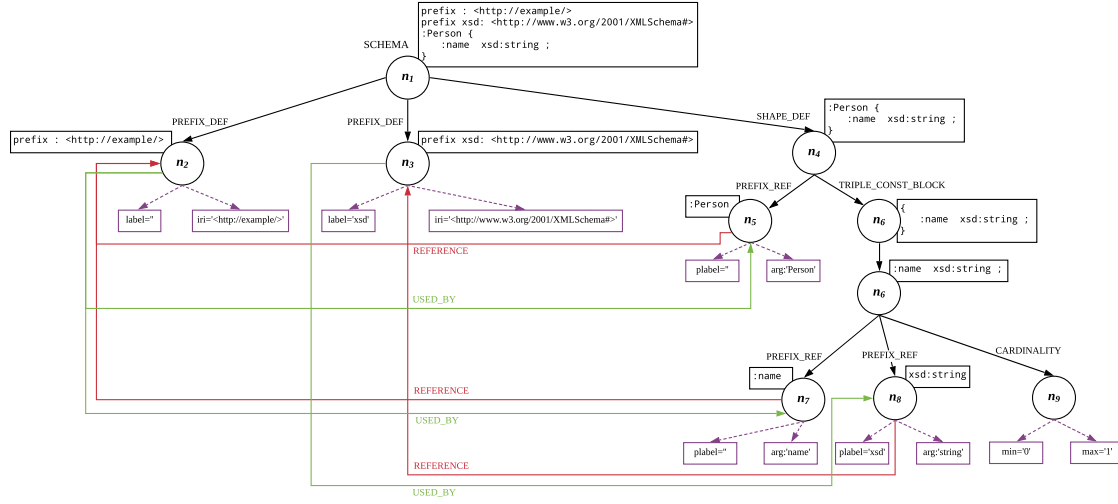
Figure 5.3: Abstract Syntax Tree produced after validation and transformations.

Shows a the resulting AST after the correspinding analysis and transformations, we call this graph the *Intermediate Language*.

Once we have the representation modeled 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 $n_{5,7\,and\,8}$ and check that there is indeed a valid reference for each of them.

Furthermore, we can even analyze 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.

5.2 Implementation

As proof of concept of the previous proposal we offer an implementation of the three components, the parser, the sintactic analyzer 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.

5.2.1 Parser

As previously discussed, the function of the parser is to extract a syntactic tree from the diagrams that we can analyze. For this purpose we decided to use the Antlr tool [27]. This tool is capable of generating sintactic analyzers from grammars defined in its own syntax. However, this tool is focused on completely processing the syntax tree and producing only

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

Figure 5.4: Checker implementation for missing semicolons warning generation.

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.

5.2.2 Sintactic Analizer

The sintactic analyzer 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 5.4 shows an example of how a sintactic 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.** 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 5.3 is very representative of this.
- **Parent node.** A pointer to the parent node.
- **Children nodes.** A list of pointers to all the children nodes.
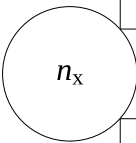
Figure 5.6 represents this information inside each node. Our default implementation only looks for the following extra syntactic pattern to the other implementations seen in Chapter 4:

```
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 5.5: Sintactic warning produced by the proposed sintactic analyzer.

Figure 5.6: Common information stored at any AST node.

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 5.5.

5.2.3   Semantic Analizer

Recall that the semantic analyzer 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 analyze 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 bellow:

- **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 subtitute 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 deafult 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 analizer 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. Some of the

5.3 Sintactic 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 analyze the rules that generate each type of event and the different error messages produced for each of them.

5.3.1 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 5.5 shows an example of this message.

5.3.2 Prefix not defined

These types of events happen when we use a referecnai 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

```
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 5.7: Semantic error produced by an undefined prefix.

```
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 5.8: Semantic error produced by an undefined shape.

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 5.7 is created.

For example in the Figure 5.3 the red lines would be the transformations done to the original AST to add the pointers to the referece nodes that point to the definition nodes.

5.3.3   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 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 5.8 is generated.

5.3.4   Prefix overriden

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

```
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 5.9: Semantic error produced by a prefix override.

```
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 5.10: Semantic error produced by a shape override.

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 5.9.

5.3.5   Shape overriden

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 5.10. If not, we will add a record to the indicated prefix scope with the corresponding information from the shape definition.

5.3.6   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 5.3.

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 5.11 is generated

```
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 5.11: Semantic warning produced by a prefix never used.

```
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 5.12: Semantic warning produced by a base set but never used.

5.3.7 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 5.12 is generated.

Part II

Translating ShEx Schemas to Object Domain Models

CHAPTER 6

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, in to object domain models. Or what it is the same, translate the ShEx schemas to objects coded in some Object Oriented Language. Figure 6.1 represents this aim. The problem is to convert the *Source* in to the *Target* (*shex → object oriented language*).

Person Schema (Source)	Person Java Object (Target)

```
1  # Prefixes...
2  :Person {
3          :name xsd:string ;
4          :knows @:Person *
5  }
```

```
1  // Imports...
2  public class Person {
3          private String name;
4          private List<Person> knows;
5          // Constructor...
6          // Getters and Setters...
7  }
```

Figure 6.1: 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 6.1, 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 PO *(Plain Objects)* [28] 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. Equation 6.1 illustrates this question where $e(x)$ measures the expressivity [29] of x.

$$e(shex\ micro\ syntax) \leq e(plain\ objects) \tag{6.1}$$

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 tripleConstraint+
7  tripleConstraint  ::= IRI_REF  constraint CARDINALITY
8  constraint        ::= IRI_REF | SHAPE_REF | "IRI" | "BNODE" |
9                        "NONLITERAL" | "LITERAL"
```

Figure 6.2: ShEx Micro Abstract Grammar.
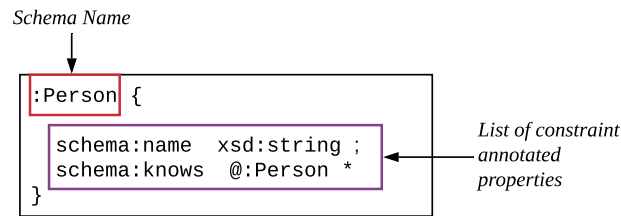
Figure 6.3: Shape expression modeling the properties of a Person.

6.1   Shape Expressions Expressivity

To measure the expressiveness of the ShEx micro syntax we will explore its abstract grammar Figure 6.2.

From the grammar we can infer that a shape is defined as an identifier and a set of triple expressions where each triple expression is in turn defined as the property, which is a reference to a prefix, the constraint and the cardinality. This, therefore, allows us to express that an object can be defined by one or by multiple properties, each with a specific type. In addition, each of these properties that define the model can be repeated n times, indicated by the cardinality. Figure 6.3 shows an example of a shape expression coded on its micro compact syntax that defines two properties for the object `:Preson`.

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 multiple cardinality so it represents a list of people you know.

So we have just seen that a shape, in its reduced grammar is a set of triple expressions made up of the property, the type and the cardinality, therefore we can formalize that a shape is defined as,

```
public class Person {

  private String name;
  private List<Person> knows;

  // Constructor
  // Getters and setters
}
```

```
@dataclass
class Person:
  name: str
  knows: List[Person]
```

```
struct Person {

  name: String,
  knows: List<Person>;
}
```

(a)   (b)   (c)

Figure 6.4: Java, Python and Rust codings of Person object. *a* corresponds to Java, *b* corresponds to Python and *c* corresponds to Rust.

$$S = \{(p_1, t_1, c_1),\ (p_2, t_2, c_2),\ \ldots,\ (p_n, t_n, c_n) \mid p \in String,$$
$$t \in RDF\ Type,\ c \in \{[0,0],\ \ldots,\ [\infty, \infty]\}\} \tag{6.2}$$

where $p \to property\ name$, $t \to type$ and $c \to cardinality$. And therefore an schema is defined as,

$$SC = \{S_1,\ S_2,\ \ldots,\ S_n\} \tag{6.3}$$

6.2 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.

6.2.1 Plain Objects Structure

From the existing programming languages we can infeer the general structure of plain objects. For this porpuse 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 6.4 shows three models that correspond to the codification of the Person schema from Figure 6.1. 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 6.5 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
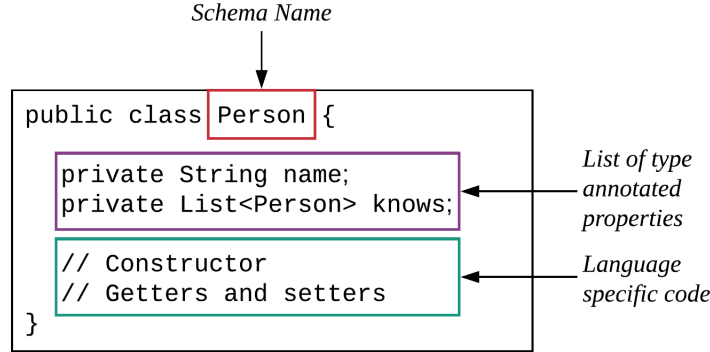
Figure 6.5: Java plain object decomposition.

corelates to the other two programming languages as they also contain this three elements. Therefore after this brief analysis we can generalize that,

$$PO = \{(p_1, t_1), \ (p_2, t_2), \ \ldots, \ (p_n, t_n) \mid p \in String,$$
$$t \in Language \ Specific \ Type \ System\} + lsc \tag{6.4}$$

where $p \to property \ name$, $t \to type$ and $lsc \to language \ specific \ code$. And therefore a model based on plain objects is defined as,

$$MO = \{PO_1, \ PO_2, \ \ldots, \ PO_n\} \tag{6.5}$$

where PO represents the plain object.

It is important to note that part of that language-specific code is responsible for representing the language-specific type system. Therefore, before formalizing our generalization, we must explore whether the type system of a language affects its expressiveness and, if so, find those types that are common to most languages of this type, if any.

6.2.2 Plain Objects Language Expressivity Dependance

In Figure 6.4 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 doesn't affect the expressivity of plain objects we will need first to find two *type-independent languages*.

Definition 6.1 (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*

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

Figure 6.7: Plain Objects Partial Generalization.

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 6.4. Rust contains the type $Either < A, B >$, 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 achive 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 assing values to Java and Rust by using the same $Either < A, B >$ type. As can be see in Figure 6.6 Java does not allow to express the same as Rust is expressing in this example. And therefore we can conclude that the expresivity of plain object is stronggly related to the build-in types that the programming language in which they are coded provides.

Person Rust Struct	Person Java Object

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

Figure 6.6: 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.

6.2.3 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 Equation 6.4 where we defined the composition of a flat object, in this way the generalization would be as indicated in Figure 6.7. As can be seen this gneralization is not complete as it does not include the production for the `type`. Thi is because we have not generalized the type system of the object oriented programing languages yet.

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

Figure 6.8: Plain Objects Complete Generalization.

However and motivated not to over-extend the scope of this work instead of extracting a generalization for the possible types that can be used in each object-oriented programming language, we will try to create this abstraction projecting it on the most common types used by XML Schema (xsd) [30]. 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 6.8 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.

6.3 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 expresivities are fully compatible or not. In Equation 6.3 we defined an schema and our aim to find if it is possible to define a function f such that $f(schema) \rightarrow model$ where $model$ is defined in Equation 6.5. To achieve this function, it is necessary that the expressiveness of the origin be less than or equal to that of the destination and now that we have already defined the expressivities of each system we can compare them.

Let's simplify the problem by eliminating all those common factors that we have in both systems. For example, from Equation 6.3 and Equation 6.5 we have lists of properties with something else, let's reduce the case to having a single property with something else like,

$$\{p,\ t,\ c \mid p \in String,\ t \in RDF\ Type,\ c \in \{[0,0],\ ...,\ [\infty,\infty]\}\};$$
$$\{p,\ t \mid p \in String,\ t \in \{String, Real, Boolean, List, Compound\}\}. \tag{6.6}$$

Also in both cases we find that the property is a free text that identifies that property, so we eliminate it from the problem. And we are left with,

$$\{t,\ c \mid t \in RDF\ Type,\ c \in \{[0,0],\ ...,\ [\infty,\infty]\}\};$$
$$\{t \mid t \in \{String, Real, Boolean, List, Compound\}\} \tag{6.7}$$

where we need to find if the existis a function f' such that we can map (t_n, c_n) to (t_n).

Now the question changes and what we have to compare is if the $RDF\ Type \cap \{[0,0],\ ...,\ [\infty,\infty]\}$ can be represented by means of $\{String, Real, Boolean, List, Compound\}$. For this purpose
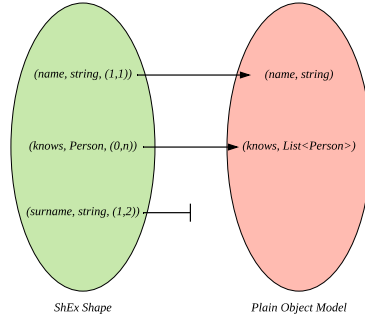
Figure 6.9: Mapping function from ShEx to Plain Object.

we will assign different values inside and outside the ranges that we have defined for each set. In addition, we will define the axiom that the cardinality $(0, \infty) \cap RDF\ Type$ is the same as $List[RDF\ Type]$. So,

$$
\begin{aligned}
(string, (1,1)) &\rightarrow & (string) \\
(Person, (0,\infty)) &\rightarrow & (List[Person]) \\
(string, (1,2)) &\rightarrow & \times \\
(Person, (2,\infty)) &\rightarrow & \times \\
(centimeters, (1,1)) &\rightarrow & \times
\end{aligned}
\tag{6.8}
$$

From the values we gave in Equation 6.8 we can see that for some cases there is no schema-object equivalence. However, for others there exist such and equivalence. Thus, we will define a Subset S' in which we will restrict the types and the cardinality so that those shapes that are defined in that subset have an equivalent object. Then,

$$
\begin{aligned}
S' = \{(p_1, t_1, c_1),\ (p_2, t_2, c_2),\ \ldots,\ (p_n, t_n, c_n) \mid p \in String, \\
t \in \{String, Real, Boolean, Compound\},\ c \in \{[1,1],\ [0,\infty]\}\}.
\end{aligned}
\tag{6.9}
$$

Thus,

$$
SC' = \left\{ S'_1,\ S'_2,\ \ldots,\ S'_n \right\}.
\tag{6.10}
$$

That means that we will not be able to translate everything that can be expressed in ShEx Micro Compact Syntax but also means that we will be able to at least translate some of the shapes. Figure 6.9 illustrates this scenario. As you can see in the figure some shapes can be mapped, like the first two, and other can't, like the third, in this case due to the cardinality not following the Equation 6.9.

CHAPTER 7

Proposed Translator

As a solution to the previous chapter, this one focuses on proposing a application $f : SC' \rightarrow MO$ such that applied on a schema, defined in Equation 6.10, results in a domain model (Equation 6.5) based on plain objects (Equation 6.4). Therefore the aim of this chapter is to once defined an schema as Equation 6.10 and a shape as Equation 6.9 define an application f such that

$$f(SC') = \begin{bmatrix} f'(S_1') \\ f'(S_2') \\ \vdots \\ f'(S_n') \end{bmatrix}, \tag{7.1}$$

where,

$$f'(p_{SC'}, t_{SC'}, c_{SC'}) = (p_{S'}, t_{S'}), \tag{7.2}$$

and give a proposition for $f'(x')$ and therefore for $f(x)$.

We know that Equation 6.9 has been obtained in such a way that $\forall \ s' \in S' \ \exists \ f : S' \rightarrow PO$ and also in Equation 6.8 we set the aggregation criterion of the cardinality component over the type component to be able to create the images of SC' in MO. Therefore f' is defined as,

$$f' : S' \rightarrow PO = f'(p_{S'}, t_{S'}, c_{S'}) \begin{cases} (p_{S'}, \ Proy_{t_{S'}} lst) & if \ c_{S'} = (1,1) \\ (p_{S'}, \ List[Proy_{t_{S'}} lst]) & if \ c_{S'} = (0, \infty) \end{cases} \tag{7.3}$$

where $Proy_{t_{S'}} lst$ represents the projection of the compatible type $t_{S'}$ from the shapes space of types over the Language Specific Types space (lst).

Therefore we not only know that it is possible to create a transformation from the subset of
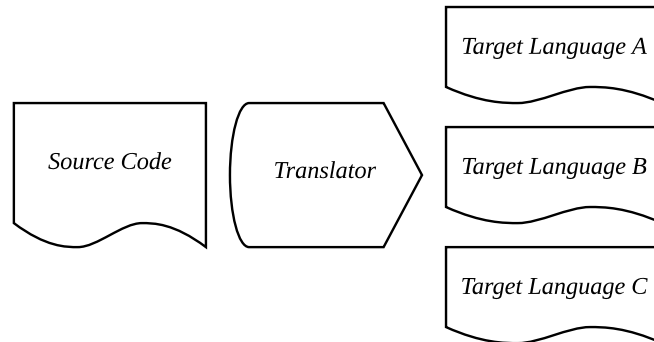
Figure 7.1: Translator generic structure.

selected schemas towards domain models formed by flat objects, but we have also identified the function that allows us to carry out this transformation. So, next we will focus on giving shape to this function so that a system can be implemented that performs the described transformations automatically.For this, in this section we will see the structure of the proposed solution as well as an implementation that allows us to validate the proposed solution to later analyze the objects generated by the solution.

7.1   Structure

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 7.1).

In addition to this, in Chapter 5, we have already developed a system capable of analyzing, 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 analyzer from Chapter 5, that makes the front end of our translator. And thenm, what we will truly develop in this section, is the back-end of the translator.

7.1.1   Translator Back-end

In our solution, the back-end of the translator, also called the synthesis phase, fulfills two main functions. On one hand, it analyzes the intermediate language in search of some specific incompatibility with the target language. And on the other hand, it generates the specific code for the target language.
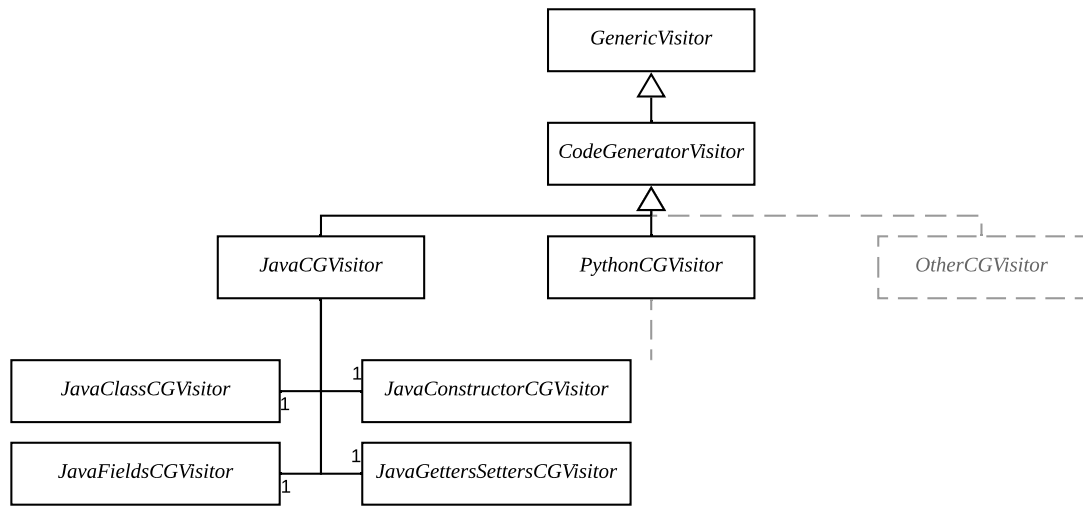
Figure 7.2: Class diagram example of the code generation visitors.

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 Equation 6.10. The visitor that is implemented is completely reused from the one developed for the Syntax or Semantic analysis in Chapter 5.

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 Equation 7.3, without prejudice to the fact that each specific code generator may have more implementations of the visitor associated. Figure 7.2 illustrates this situation where the Java code generartion visitor actually contains four more visitors one for each language specific level of the plain objects.

7.2   Implementation

7.3   Generated Obejcts

<div style="display:flex">

```
            Person.shexl
11   # Prefixes...
12   :Person {
13          :name xsd:string ;
14          :knows @:Person *
15   }
```

```
                Person.java
16   // Imports...
17   public class Person {
18          private String name;
19          private List<Person> knows;
20          // Constructor...
21          // Getters and Setters...
22   }
```

</div>
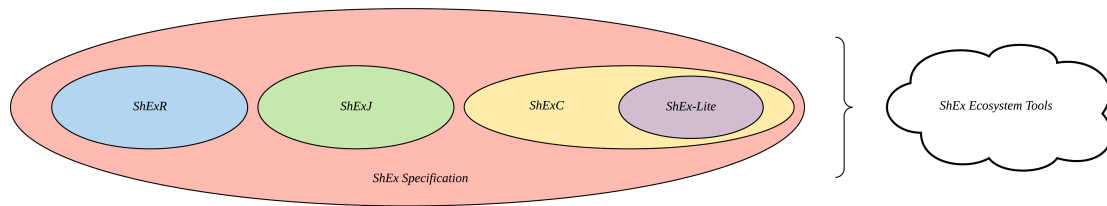
Figure 7.3: Mental model of ShEx-Lite in the existing ShEx syntaxes context. From this model we can see that Shex-Lite is in fact an strictly subset of ShExC, which follows the ShEx Specification. And therefore ShEx-Lite will also follow that expecification, which automatically enables ShEx-Lite schemas to be used in any other existing ShEx tool.
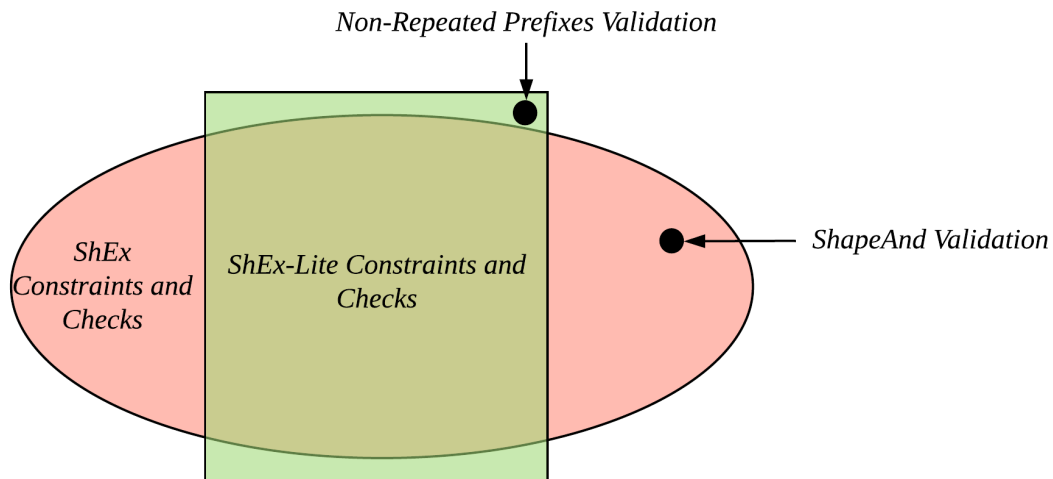
Figure 7.4: Constraints and checks context diagram for ShEx-Lite and ShEx.

Figure 7.5: Schema modeling a `Person` in `shexl` syntax to the left. And the `ShEx-Lite` generated code in `Java` to the right.

Part III

Project Sinthesis

CHAPTER 8

Evaluation of Results

8.1 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.

8.2 Dataset

The previous methodologies will be used on a dataset of its own shape expressions. As it does not currently exist, to the best of our knowledge, no dataset of shape expressions has been used as a sample GitHub. On this platform we have collected all files with the `.shex` extension and licensed for use. In addition, we have filtered and reduced to only those schemas that were expressed through ShEx's reduced grammar.

8.3 Results

CHAPTER 9

Planning and Budget

9.1 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 defense of the work**. Therefore planning revolves around these milestones *(IDs 3, 4 and 5 from Figure 9.1)*.

9.1.1 Presentation of the Proposal

The acceptance of the proposal includes the first tasks *(IDs 6-8 from Figure 9.1)* 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.

9.1.2 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 9.1)* of the work, in our case they are to carry out the corresponding research to understand the scope of the proposed objectives, to propose a solution and to obtain a few relustados 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.

9.1.3 Defense of the Work

The defense of the project corresponds to those tasks *(IDs 34-36 from Figure 9.1)* subsequent to the delivery of the dissertation and that have to do with public defense in which the work carried out is evaluated.

So as you can see the main project statistics are shown in Table 9.1.

Table 9.1: Statistics of the main project tasks.

Phase	Duration
Proposal Preparation	*3.5 days / 14 h.*
Dissertation	*94 days / 376 h.*
Viva Voice	*4 days / 16 h.*
Total	*105 days / 420 h.*

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 Sintactic and Semantic Analyzers	4 days	Wed 17/06/20	Mon 22/06/20
28		28	1.5.5		Proposed Sintactic 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

Page 1

Figure 9.1: Tasks planning of the project.

9.2 Budget

CHAPTER 10

Conclusions

10.1 Future Work

Currently both proposed solutions are based on the reduced ShEx grammar, therefore the first future work we identify is to be able to bring the philosophies described in this work to the full ShEx grammar, so that the improvements described can benefit all users of the language.

The next step would be to expand the range of the static analysis of shape expressions so that it supports more elements of the grammar so that all the elements that make up a shape, their dependencies and relationships can be analyzed in much more detail.

Part IV

Annexes and References

APPENDIX A

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-\uEFFFF] ;
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]+ ;
```

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
16   STRING_LITERAL:          STATIC_STRING_LITERAL;          // Meant to be extended with interpolated text
17   STATIC_STRING_LITERAL:   '"' Quoted_text? '"';
18   IRI_LITERAL:             '<' (~[\u0000-\u0020=<>"{}|^'\\] | Unsigned_character)* '>';
19   DECIMAL_LITERAL:         ('0' | [1-9] (Digits? | '_'+ Digits)) [lL]?;
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\\tnr"']
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 Hexadecimal_digit
95    ;
96
97   fragment Unsigned_character
```

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

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? SEMI?
93    ;
94
95   constraint_node_iri_expr
96    : IRI
97    ;
98
99   constraint_valid_value_set_type
100   : call_prefix_expr
101   | call_shape_expr
102   | literal_string_value_expr
103   | literal_real_value_expr
104   ;
105
106  constraint_node_any_type_expr
107   : ANYTYPE
108   ;
109
110  constraint_node_non_literal_expr
111   : NONLITERAL
112   ;
113
114  constraint_value_set_expr
115   : LBRACK constraint_valid_value_set_type* RBRACK
116   ;
117
118  constraint_node_bnode_expr
119   : BNODE
120   ;
121
122  constraint_node_literal_expr
123   : LITERAL
124   ;
125
126  call_expr
127   : call_prefix_expr
128   | call_shape_expr
129   ;
130
131  call_prefix_expr
132   : pref_lbl=IDENTIFIER? COLON shape_lbl=IDENTIFIER
133   | base_relative_lbl=literal_iri_value_expr
134   ;
135
136  call_shape_expr
137   : AT prefix_lbl=IDENTIFIER? COLON shape_lbl=IDENTIFIER
138   | AT base_relative_lbl=literal_iri_value_expr
139   ;
```

APPENDIX C

Project Communications

C.1 Open Source Community

Regarding the open source community, from the beginning the work was considered as a collaborative project where the community could debate, validate and contribute new ideas to the project.

The http://github.com/weso/shex-lite repository has primarily been used as the central source of code. But there is a parallel repository http://github.com/weso/shex-lite-evolution where there are records of some proposals that affect the design of the system.

GitHubFlow[1], a variant of GitFlow[2] oriented to unlock the advance that occurs on many occasions, has been used as a methodology to work. In this way a user can send an issue, make the appropriate code modification and create a pull request that once accepted becomes directly part of the most recent version of the system. An example of this is the pull request SLP-0143 https://github.com/weso/shex-lite/pull/143 where a community user implemented python code generation on their own.

C.2 Scientific Disclosure

The research work of this dissertation has lead to papers that has been sent to different conferences. The following papers are somehow derived from this dissertation:

1. ShEx-Lite: Automatic generation of domainobject models from Shape Expressions. Guillermo Facundo Colunga, Alejandro González Hevia, Jose Emilio Labra Gayo, and Emilio Rubiera Azcona. *19th International Semantiuc Web Conference. Posters and Demos Track.*

---

[1] https://guides.github.com/introduction/flow/

[2] https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

C.3   Community Meetings

Also framed in the project, various meetings have been held with entities such as the Dublin Core Metadata Initiative (https://www.dublincore.org/), Eric Prud'hommeaux (father of ShEx) or the management office of the European Hercules ASIO project. During the Erick meeting the concept of the ShEx micro compact syntax and its Antlr transformation where disscussed. During the DCMI meetings we discussed the aims of the project and they validate them. And with the ASIO management office we disccussed how they will adopt out proposed solution as a production system for generating plain objects from their schemas.

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] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, 1(1):7–87, 1990.

[9] List of object-oriented programming languages, April 2020. Page Version ID: 950660258.

[10] 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.

[11] Programming language, May 2020. Page Version ID: 958722580.

[12] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.

[13] Tim Berners-Lee et al. Semantic web road map, 1998.

[14] Eric Van der Vlist. *Relax ng: A simpler schema language for xml.* " O'Reilly Media, Inc.", 2003.

[15] Domain-specific language, March 2020. Page Version ID: 945660040.

[16] Turing completeness, May 2020. Page Version ID: 954916159.

[17] Lexical analysis, May 2020. Page Version ID: 955735363.

[18] Optimizing compiler, April 2020. Page Version ID: 949660274.

[19] Entorno de desarrollo integrado, May 2020. Page Version ID: 126109294.

[20] dotnet/roslyn. Library Catalog: github.com.

[21] 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.

[22] Iovka Boneva, Jérémie Dusart, Daniel Fernández Alvarez, and Jose Emilio Labra Gayo. Semi automatic construction of shex and shacl schemas. 2019.

[23] Daniel Fernández-Alvarez, Jose Emilio Labra-Gayo, and Herminio Garcıa-González. Inference and serialization of latent graph schemata using shex, 2016.

[24] Herminio Garcia-Gonzalez, Daniel Fernandez-Alvarez, and Jose Emilio. Shexml: An heterogeneous data mapping language based on shex.

[25] Robert W Floyd. Syntactic analysis and operator precedence. *Journal of the ACM (JACM)*, 10(3):316–333, 1963.

[26] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.

[27] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[28] Martin Fowler. *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.

[29] Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.

[30] Xsd simple elements.