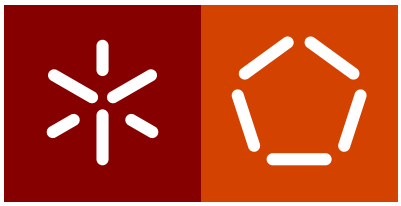
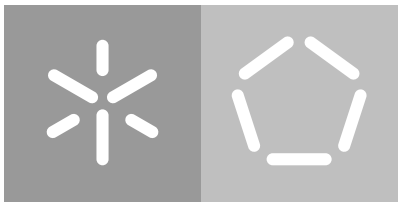


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo André Coelho Cardoso

Synthetic Data Generation from JSON/XML Schemas

Outubro 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo André Coelho Cardoso

Synthetic Data Generation from JSON/XML Schemas

Master's Dissertation
Integrated Master's Degree in Informatics Engineering

Dissertation supervised by
José Carlos Leite Ramalho

Outubro 2022

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use or falsification of information or results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Assinado por: **Hugo André Coelho Cardoso**
Num. de Identificação: 15619313
Data: 2022.10.12 18:45:17+01'00'

ABSTRACT

The objective of this dissertation is the development of an application capable of automatically generating synthetic datasets that are representative and, possibly, very large, directly from *JSON* and *XML* schemas, in order to facilitate the testing of software applications and scientific endeavors in areas such as Data Science or Application Development.

For this purpose, it is intended to develop a new version of *DataGen*, an online open-source application that allows the quick prototyping of datasets through its own *Domain Specific Language* (*DSL*) of specification of data models. *DataGen* is able to parse these models and generate synthetic datasets according to the structural and semantic restrictions stipulated, automating the whole process of data generation with spontaneous values created in runtime and/or from a library of support datasets.

The objective of this new product, *DataGen From Schemas*, is to expand *DataGen*'s use cases and raise the datasets specification's abstraction level, making it possible to generate synthetic datasets directly from schemas. This new platform builds upon its prior version and acts as its complement, operating jointly and sharing the same data layer, in order to assure the compatibility of both platforms and the portability of the created *DSL* models between them. Its purpose is to parse schema files and generate corresponding *DSL* models, effectively translating the *JSON* or *XML* specification to a *DataGen* model, then using the original application as a middleware to generate the final datasets.

The present dissertation details the entire creative process behind the development of this application: firstly, it frames the topic of study and its initial phase of investigation, debating relevant technologies and existing related work; then, the ideation phase of the product is addressed, projecting an adequate arquitecture and the reasons behind its design choices, as well as surveying technical requirements for *DataGen From Schemas*, while taking into account the conclusions reached through prior research; afterwards, the development phase is covered, carefully explaining the elaborated components, their properties and the data flow between them, for both the *JSON* and *XML* modules; finally, the reader is presented with conclusions taken from this project's development and possible future work to implement, in order to improve the current solution.

Keywords: Schemas, JSON, XML, Data Generation, Synthetic Data, DataGen, DSL, Dataset, Grammar, Randomization, Open Source, Data Science, REST API, PEG.js

RESUMO

O objetivo desta dissertação é o desenvolvimento de uma aplicação que permita gerar automaticamente *datasets* sintéticos representativos e possivelmente extensos, a partir de *schemas* de *JSON* e *XML*, de forma a facilitar a testagem de aplicações de *software* e empreendimentos científicos em áreas como a Ciência de Dados e o Desenvolvimento de Aplicações.

Para esta finalidade, pretende-se desenvolver uma plataforma assente sobre o *DataGen*, uma aplicação que permite a prototipagem rápida de *datasets* através da sua própria *Domain Specific Language* (*DSL*) de especificação de modelos de dados. O *DataGen* é capaz de processar estes modelos e gerar posteriormente *datasets* sintéticos que obedeçam às restrições estruturais e semânticas estabelecidas, automatizando todo o processo de geração de dados com valores espontâneos gerados em tempo de execução e/ou provenientes de bancos de dados de suporte.

O objetivo deste novo produto, *DataGen From Schemas*, é expandir os casos de aplicação do *DataGen* e aumentar o nível de abstração da especificação de modelos de dados, tornando possível a geração de *datasets* sintéticos diretamente a partir de *schemas*. Esta nova plataforma estará assente sobre a sua versão anterior e agirá como seu complemento, operando conjuntamente e partilhando a mesma camada de dados, de forma a assegurar a compatibilidade das plataformas e a portabilidade dos modelos criados entre ambas. O seu propósito é processar ficheiros *schema* e gerar modelos correspondentes na *DSL*, efetivamente traduzindo a especificação em *JSON* ou *XML* para um modelo do *DataGen*, para depois usar a aplicação original como um *middleware* para gerar os *datasets* finais.

A presente dissertação detalha todo o processo creativo por detrás do desenvolvimento desta aplicação: começa por enquadrar o tema de estudo e a sua fase inicial de investigação, debatendo tecnologias relevantes e trabalho relacionado existente; de seguida, é abordada a fase de ideação do produto, projetando uma arquitetura adequada para a solução e as razões por detrás das suas escolhas de design, e realizado um levantamento de requisitos técnicos para o *DataGen From Schemas*, tendo sempre em conta as conclusões alcançadas através de investigação prévia; depois, é relatada a fase de desenvolvimento do produto, explicando minuciosamente os componentes elaborados, as suas propriedades e o fluxo de dados entre eles, para ambos os módulos de *JSON* e *XML*; finalmente, são apresentadas ao leitor as conclusões retiradas do desenvolvimento deste projeto e possível trabalho futuro a implementar, de forma a melhorar a solução atual.

Palavras-Chave: Schemas, JSON, XML, Geração de Dados, Dados Sintéticos, DataGen, DSL, Dataset, Gramática, Aleatoriedade, Open Source, Ciência de Dados, REST API, PEG.js

CONTENTS

LIST OF FIGURES

ACRONYMS

A

API Application Programming Interface.

C

CPU Central Process Unit.

D

DFJS DataGen From JSON Schemas.

DFXS DataGen From XML Schemas.

DOM Document Object Model.

DSL Domain Specific Language.

H

HTTP Hypertext Transfer Protocol.

I

IOT Internet of Things.

IT Information Technology.

J

JSON JavaScript Object Notation.

JWT JSON Web Token.

L

LCM Least Common Multiple.

M

MVVM Model-View-ViewModel.

N

NOSQL Not Only SQL.

P

PHD Doctor of Philosophy.

R

REST Representational State Transfer.

RNID National Digital Interoperability Regulation.

S

SLATE Symposium on Languages, Applications and Technologies.

SPA Single-Page Application.

U

URI Uniform Resource Identifier.

W

W3C World Wide Web Consortium.

X

XML Extensible Markup Language.

XPATH XML Path Language.

XSD XML Schema Definition.

1

INTRODUCTION

This introductory chapter's purpose is to establish the motif of the present dissertation, contextualizing it in the current panorama of software development and explaining its necessity and possible application in a wide array of scientific areas. The motivation behind the development of this project is then explained, along with the objectives it aims to fulfill and the methodology under which it will unroll, concluding with a brief summary of the structure of this dissertation and the topics addressed in each chapter.

1.1 CONTEXT

The current landscape of the software development market is being increasingly occupied with scientific areas that operate with large amounts of data. A prime example is that of Data Science, which aims to apply methods of scientific analysis and algorithms to bulky datasets, in order to try to extract knowledge and conclusions from the available information, which may then be used for several other means. Another case is that of Machine Learning, which looks to use this method to equip informatic systems with the capacity of self-learning, accessing data and learning from its analysis, as to become more efficient in their function.

However, in a world where the need for bulky and representative datasets increases by each passing day, data privacy policies and other concerns related to the privacy and safety of users (Mostert et al., 2016) present themselves as difficult obstacles to the growth of these sciences and to the progression of many projects in the development (GAO, 2020) and testing phases. Moreover, the inability to share data which stems from such concerns also frequently prevents companies from seeking outside help and/or collaboration (Patki et al., 2016).

In this context, the generation of synthetic data arises as a possible solution, under the premise of being able to create realistic, large datasets artificially, from the given structural specifications. This approach was originally proposed by Rubin in 1993 (Rubin, 1993), as an alternative that made it possible to use and share data without disrespecting the present rigorous regulations regarding the handling of sensitive data (such as the European Union's GDPR (GDP, 2018)), since the data in question, altough very similar to corporate datasets concerning real users, would not be obtained by direct measurement. Since then, this method

has been further explored and refined, being applied in the most diverse areas such as Smart Homes (Dahmen and Cook, 2019), spacial microsimulation models (Smith et al., 2009), *Internet of Things* (*IoT*) (Anderson et al., 2014), Deep Learning, (Ekbatani et al., 2017) and automotive applications (Tsirikoglou et al., 2017).

The present dissertation serves to expose and document the processes of ideation and development of the application *DataGen From Schemas*, whose objective is to generate synthetic datasets directly from either *JSON* or *XML* schemas. This application must fulfill two requisites: on the one hand, it must be able to generate datasets of ample size and with realistic information; on the other hand, it must also be able to parse the users' schemas and obey their specifications, so that the created data is formally and structurally compliant. In order to satisfy these conditions, the platform will be built upon another existing application, *DataGen*, that will be contextualized in chapter 2.

1.2 MOTIVATION

DataGen is a versatile tool that allows the quick prototyping of datasets and testing of software applications. Currently, this solution is one of the few available that offers both the complexity and the scalability necessary to generate datasets adequate for demanding tasks, such as the performance review of data *API*s or complex applications, making it possible to gauge their ability to handle an appropriate volume of heterogeneous data.

The core of *DataGen* is its own *Domain Specific Language* (*DSL*), which was created for the purpose of specifying datasets, both at a structural level (field nesting, data structures) and a semantic level (values' data types, relations between fields). This *DSL* is endowed with a wide range of functionalities that allow the user to specify different types of data, local relations between fields, the usage of data from support datasets depicting different categories, the structural hierarchy of the dataset and many other relevant properties, as well as powerful mechanisms of repetition, fuzzy generation, etc. This allows for the generation of very miscellaneous and representative datasets in either *JSON* or *XML*, while dealing with intricate and demanding requirements.

In the current scheme of software development, any enterprise that operates with *JSON* or *XML* data must have well-defined and thorough formal specifications to control and monitor the data flow in their applications, in order to assure that incoming information is well-structured and compliant with the software's requirements and outgoing data is presented as intended and does not produce unexpected behaviour. As such, it is essential to formulate schemas for semantic and structural validation, by modeling data either internally or via third parties with tools oriented to this goal, in order to restrict and enforce the content intended for each solution.

Considering the present necessity for these schemas in enterprises' business models and their recurring usage, it stands to reason that a program capable of producing large and accurate datasets from *JSON*/*XML* schemas is an incredibly valuable asset, as it provides the capacity to quickly and effortlessly create representative data to test and debug platforms in development, as well as to evaluate their performance under heavy stress, without having to manually concoct the information or wait for third parties to provide such resources.

As such, *DataGen From Schemas* emerges as a complement and an extension to its prior version, looking to generate datasets directly from schemas. By doing this, the user is given the option to specify the structure of the intended dataset in either *JSON* or *XML* Schema. This arises as an alternative to the definition of the operational rules of the dataset in *DataGen*'s native *DSL*, for which the user must first learn how to use it, through the lengthy documentation available. With this, the formulation of the *DSL* model becomes an intermediate step executed in the background and the user only has to interact with the schema and the resulting dataset. However, the generated *DSL* model will also be made available, to enable further customizability in *DataGen*.

As such, this new product aims to offer a solution for a present and generalized need in the software development process and increase *DataGen*'s use cases significantly, making the dataset generation process simpler and more acessible to any user. This new component acts as an abstraction layer over the existing application, ignoring the necessity to learn how to use the *DSL* from its documentation and greatly expediting the process of structural specification of the dataset.

1.3 Objectives

The main goal of this dissertation is the generation of datasets from schemas (formal specification files) written in either of the data exchange formats *JSON* and *XML*. More specifically:

- Development of a compiler to filter and preprocess *JSON* schemas and another for *XML* schemas;
- Creation of converter programs capable of generating valid *DataGen DSL* models from the filtered data;
- Development of a web application, with user-friendly features and an intuitive worflow;
- Creation of *REST* routes to enable the application's usage without its interface and eventual third party integration;
- Establishment of communication between the new application and its prior version;

- Testing of the implementation;
- Deployment of *DataGen From Schemas*.

1.4 Methodology

The work methodology for the present dissertation follows these steps:

- Bibliographical research on synthetic data generation from schemas;
- Requirements elicitation and analysis for the application;
- Study and decision of adequate technologies for the implementation;
- Development of the solution;
- Testing and performance analysis of the implementation;
- Weekly meetings with the supervisor.

1.5 Structure of the Document

The present chapter's purpose was already established in the beginning of the chapter.

Chapter 2 describes the state of the art, i.e. it presents all relevant knowledge obtained on the topic of this dissertation, relative to pertinent technologies and existing related work. It begins with a study of *JSON* and *XML*, in order to gain a clear understanding of what is intended to generate and what difficulties may originate from the differences between these two formats. Next, a deep analysis is carried out on *DataGen*, the application that will act as the foundation of this new version, to show in what measures it is adequate for this role and determine what shortcomings it has that will need to be addressed. Finally, several study cases of similar existing applications are carefully investigated with the objective of identifying their advantages and weaknesses, in order to perform a minute elicitation of requirements for *DataGen From Schemas*.

Afterwards, in chapter 3, a detailed and well-substantiated approach is proposed for the development of the final product, postulating a compartmentalized architecture that shares certain layers/components with *DataGen*, justifying the technologies of choice and clearly outlining what functionalities are meant to be implemented in each part of the application.

Chapter 4 covers everything that was developed in the scope of this project, starting with a meticulous explanation of both the *JSON* Schema and *XML* Schema components of the product, namely their respective modules and workflow, problems faced during their implementation and the approaches chosen to solve them, both from an infrastructural and

an algorithmic standpoint, then an exposition of other extra features developed for user conveniency and, finally, an overview of the application's graphical interface.

Chapter 5 displays several complex example use cases that well demonstrate the capabilities and efficiency of *DataGen From Schemas*, which were used to test the application.

Finally, chapter 6 presents conclusions on the work performed and reported in this thesis, contemplating the final product that was accomplished and comparing it to the software ideated in earlier stages, identifying its biggest successes and existing flaws and anticipating future work that could be done to improve the application.

2

STATE OF THE ART

This chapter exposes all knowledge obtained from the research carried out, within the scope of this project, on the central themes of the present dissertation and related work that is currently available. Firstly, the *JSON* and *XML* formats are subjected to a detailed study, in order to investigate what kind of information each of them can represent and in what circumstances they should be utilized, so as to have a clear understanding of the data to be produced. Following this, a minute analysis is carried out on *DataGen*, the base application upon which *DataGen From Schemas* will be constructed, which aims to examine its compatibility with schemas and determine which of its functionalities will be useful for the schema translation procedure. Lastly, related work available online is carefully studied - in this case, *JSON* and *XML* generators from the respective schemas. The goal is to perform a thorough requirements elicitation, dissecting the existing products to identify their advantages and weaknesses, in order to objectively determine lists of features that are desirable and others to avoid in *DataGen From Schemas*.

2.1 FORMATS TO ADOPT

The portuguese *National Digital Interoperability Regulation* (*RNID*), published in 2018, is a regulation the aims to standardize the provision of information in the web performed by *IT* systems in Portugal, establishing that all existing information in circulation must be provided in open, non-proprietary formats, in order to assure the technical and semantical interoperability, in general terms, within the Public Administration. This norm, which reflects a growing universal approach to online data exchange (as reflected by the european guidelines to interoperability, for example), resulted in the majority of portuguese websites leaning towards two data formats in particular, traditionally used for communication between web services - *JSON* and *XML*.

In historical terms, *XML* was initially the most popular format for data exchange between applications, during the emergence and establishment of web services. Over time, it was concluded that the format was too heavy for this purpose, as it was observed that the data files were too large and delayed the communication between applications and even

their very own provision of services. As such, new alternatives started being explored, namely *JSON*, which turned out to be one of the most interesting and practical solutions, thanks to its lightweight and compact nature, reason for which it started being incrementally implemented in this area over the following years. Ultimately, with the arrival of *REST* architectures, *JSON* became the format of election for data exchange between web services, even though *XML* still continues to be widely used nowadays in this context.

As such, the data formats that *DataGen From Schemas* aims to implement are precisely *JSON* and *XML*, which, even though they are considered competitors in the context of data exchange, are able to represent different types of information and have distinct applications. This section will conduct an analysis of these formats, in order to better understand their structures, identify the main differences between these two languages and foresee what obstacles may arise in their generation through *DataGen*, which will be discussed in further detail in section 2.2.

2.1.1 *JSON*

JavaScript Object Notation (*JSON*) is a lightweight data-interchange format derived from the programming language *JavaScript*, whose simplicity and readability contributed to its current wide popularity. It possesses an ample and growing support in numerous programming languages (*C++, Java, JavaScript, Python*, etc), presenting itself as a strong candidate for fast and compact exchange of information between applications.

This format is used to represent **structured information**, i.e. data with rigid and well-defined configurations, where no divergencies are allowed from the structure established in the schema, e.g. a different data type for a certain field. *JSON*'s syntax reflects this trait, having only four different primitive types for values - string, number, boolean and null - and constructing instances entirely upon only two different data structures: objects (non-ordered collections of key/value pairs) and arrays (ordered lists of values).

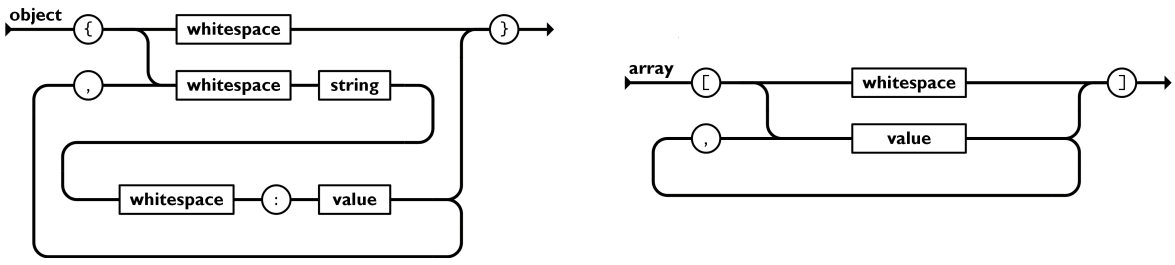


Figure 1: Structures of a JSON object and array.

2.1.2 *XML*

Extensible Markup Language (*XML*) is a flexible text-based meta-language, i.e. a markup language usable on its own, but that can also act as a basis for other languages, for publishing and exchange of information.

This format supports both **structured** and **semi-structured data** - data whose model determines a flexible formal structure, which can provide several degrees of freedom, while still having markers (in this case, *tags*) to separate structural elements and distinguish different fields in the instance. Below is an example of semi-structured data in *XML*, with the respective specification in *XML Schema Definition* (*XSD*):

```
<xs:element name="paragraph">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="name" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="year" type="xs:string"/>
      <xs:element name="profession" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

```
<paragraph>
    <name>Albert Einstein</name> (<year>1879</year>-<year>1955</year>) was a
    german <profession>theoretical physicist</profession> born in the city of
    <city>Ulm</city> in <year>1879</year>. In his adolescence,
    <name>Einstein</name> moved to <country>Switzerland</country>, where
    he graduated, ultimately becoming a <profession>professor</profession> in
    the Swiss Federal Institute of Technology in <country>Zürich</country>.
</paragraph>
```

Figure 2: Definition of semi-structured data in *XML* Schema and respective instance.

Moving on to the study of the *XML* syntax and the structure of the resulting instances, it is possible to observe the following characteristics:

- usage of **tags** to separate the content from the formatting, clearly outlining where the content of each element begins and ends;
- **hierarchical structure of elements** - for each record, there is a root element and all others are nested in it;
- each element possesses:

- **opening and closing tags**, or only one tag, optionally, if the content is empty;
- **attributes**: these represent "properties" of the element, information complementary to the content;
- **content**: can be either simple (text), complex (nesting of other elements) or mixed (both).

2.2 DATAGEN

DataGen is a full-stack web application that is open-source and free to use, available at `https://datagen.di.uminho.pt/`. It was developed by the author of the present dissertation, Hugo Cardoso, in collaboration with other colleagues and under the guidance of the advisor José Carlos Ramalho, who posteriorly proposed a follow-up project to expand the application's functionalities, for it to be able to generate datasets directly from schemas. This idea would be further refined and ultimately become the central topic of this dissertation, materialized in *DataGen From Schemas*.

It is strongly encouraged to check the published paper on *DataGen*'s development and functionality first (Santos et al., 2021), in order to be better contextualized in the capacities of this product and have a greater understanding of what will serve as the foundation for the new software described. For the sake of brevity, *DataGen*'s *DSL* mechanisms will not be explained in this document and it will be assumed that the reader is familiar with them, going forward.

DataGen is a powerful and versatile tool that can generate large datasets with diverse content according to the user's needs, from specifications written in its own *DSL*. For it to serve as the foundation of *DataGen From Schemas*, it is necessary to ensure that this application possesses the means necessary to generate data from schemas, i.e. to verify that there is compatibility between the information and structures specified in schemas and the program's method of generation and functionalities. The objetive is to use *DataGen* as a middleware: since this application is already capable of generating datasets from *DSL* models, there is no need to reinvent the wheel - the intent is to create a preprocessing routine capable of parsing schemas and translating them to *DSL* models, then charging *DataGen* with producing the final result from these intermediate models. As such, *DataGen* must fulfill several requirements:

1. Capacity to generate data in either *JSON* or *XML*;
2. Capacity to generate both structured (*JSON*) and semi-structured (*XML*) information;
3. Capacity to program every primitive data type made available in schemas;
4. Capacity to generate recursive structures;

5. Capacity of fuzzy generation.

2.2.1 *Generation in JSON and XML*

DataGen provides the option to generate datasets in both *JSON* and *XML*. During the parsing of the *DSL* model, its compiler creates an intermediate data structure with the final dataset, correspondent to a *JSON* object, which can then be translated, or not, to *XML* (Santos et al., 2021). This translation procedure creates a string with data in *XML*, as it iterates the structure recursively.

Thus, it can be concluded that the translation from *JSON* Schema to *DataGen*'s *DSL* and from the *DSL* to *JSON* is straightforward and without any major complications, given that the intermediate data structure generated by the program is also a *JSON* object. Furthermore, *JSON* and *JSON* Schema share the same syntax, since the latter is a *JSON* vocabulary, which means it is written in *JSON* and operates under a strict set of rules, where specific keywords have precise meanings and can be used to annotate or validate *JSON* documents. This common syntax makes it easy to interpret the schema and validate the instance, resulting in a direct translation along the aforementioned pipeline. As such, *DataGen* is ideal for creating datasets with structured information in this data format.

However, the application's generation procedure of *XML* data is rudimentary: it only produces elements, possibly nested, with an opening tag, the respective value and a closing tag. This is because the software was originally projected to fabricate data in *JSON*, hence the syntax of the *Domain Specific Language* being inspired in this format, having only later in the development phase been added the feature of translation to *XML*. This key detail entails complications for the translation of *XML* Schema to the *DSL*, due to a crucial set of structural differences between the *JSON* and *XML* formats, which originates from the following *JSON* traits:

- it has no notion of attributes;
- defines only structured content and is not able to represent semi-structured information, unlike *XML*. *JSON* is not capable of wrapping a given property with text, since that text itself would also need to be formatted as a key/value pair in order to be accepted in the *JSON* structure;
- does not support several properties having the same key, at the same depth in the structure - this results in *JSON* overwriting them in the instance and only saving the final value declared, meanwhile this is possible in *XML*.

In order to circumvent these issues, it will be necessary to expand the funcionalities of *DataGen*'s translator to *XML*. Under the aforementioned circumstances, the *DSL* model

produced by *DataGen From Schemas* will be annotated with special flags in the properties' keys, which will be posteriorly parsed by the *XML* translator, as a means to bypass the gaps in *JSON*'s ability to represent *XML* information. The translator will then effect the required changes to the elements according to each flag's meaning, before removing it and preserving only the original key, presenting a clean and formatted final dataset to the user. *DataGen From Schemas* will follow a difficult to read and relatively elaborated convention for these markers, in order to ensure that users of *DataGen*'s original version don't use them by coincidence while constructing their *DSL* models and obtain results different from the intended. This strategy should require at least the following markers, whose implementation will be further discussed in 4.2:

  - attribute: $|\mathrm{DFS}_{ATTR_{attribute_{name}}}|;semi-structuredelement:|DFS_{MIXED_{key}}|;$

- the elements' keys will be numbered, keeping a separate counter of occurrences for each distinct key, in order to not overwrite them in the intermediate *JSON* structure: DFS_1__{key}, DFS_2__{key}, ...

With this method, it becomes possible to represent semi-structured content in the generated data, since the translator to *XML* will take charge of identifying the elements with mixed content, through the respective flag, and wrapping them in *lorem ipsum*. Requirement 2 is thus fulfilled.

As such, the translation routine will become much richer in terms of functionalities and it will be possible to generate much more complex and interesting datasets in *XML*, which also checks requirement 1 in its entirety.

2.2.2 *Generation of Primitive Data Types*

Besides their structural differences, *JSON* and *XML* schemas also possess distinct primitive data types. To make sure that *DataGen* is an adequate application upon which to base this project, it is necessary to guarantee that it is possible to generate each of those types through the tools of its domain language.

Furthermore, the ability to induce controlled randomness in the values' generation procedure is also pertinent and very relevant, so that the result is not deterministic. By using *DataGen* as a middleware, the intent is to provide users with the intermediate *DSL* model as well, giving them the option to utilize it directly in *DataGen*'s original version (either via its front end or by *HTTP* request), if they so choose, and customize it to their liking. As such, *DataGen From Schemas* should be able to introduce controlled randomness in the values of the model it generates, so that it does not always produce the same instance whenever it is parsed, thus allowing to generate heterogeneous datasets from the same initial state.

JSON Schema

The main component of *DataGen* is its *Domain Specific Language* (*DSL*), which allows the user to specify the intended dataset, both at a structural level (field nesting, data structures) and a semantic level (values' data types, relations between fields). Its syntax is built upon the *JSON* format, having been incrementally attached with a wide range of functionalities, namely mechanisms of repetition, fuzzy generation, support datasets, among others. As such, there is a direct translation from *JSON* to the *DSL*, so it already incorporates all of *JSON*'s primitive data types:

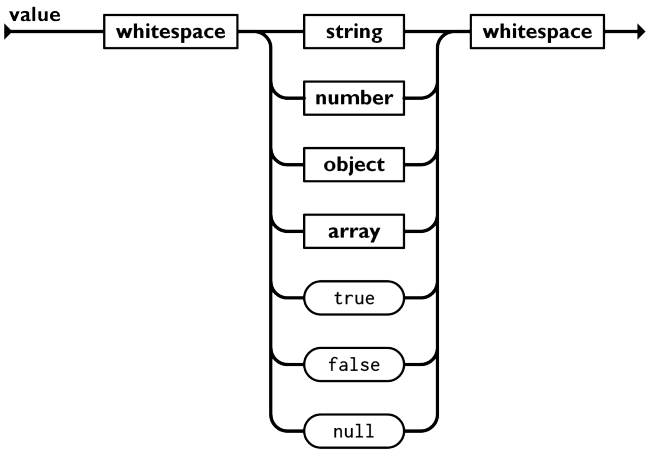


Figure 3: Primitive data types of the *JSON* format.

Below is presented a simplistic example of these types' representation in *DataGen*'s *DSL*:

```
{
  boolean: true,
  integer: 14,
  number: 54.832,
  null: null,
  string: "example",
  array: [ 1, "2", 3, "four" ],
  object: {
    animal: "Whale"
  }
}
```

Figure 4: Simplistic example of *JSON* data types in *DataGen*'s *DSL*.

Therefore, it is possible to generate any primitive data type programmable in *JSON* Schema. However, the code snippet presented above is deterministic, so it is not ideal yet.

It is here that *DataGen*'s **interpolation functions** come in, a powerful tool of its grammar that allows the user to introduce premeditated randomness in the values' generation, by restricting their type and possible range of values. The application has a vast documentation of all the functions made available and how they operate, listing every data type that can possibly be generated and the restrictions (arguments) appliable to each of them. Rewriting the above example with interpolation functions, it is possible to obtain the following model, which can produce a huge combination of different results:

```
{
  boolean: '{{boolean()}}',
  integer: '{{integer(0, 1000)}}',
  number: '{{float(-100, 500.3)}}',
  null: null,
  string: '{{lorem('words', 5, 10)}}'
  array: [ 1, "2", 3, "four" ],
  object: {
    animal: '{{animal()}}'
  }
}
```

Figure 5: Example of *JSON* data types with interpolation functions in *DataGen*'s *DSL*.

Thus, it can be concluded that *DataGen* can not only generate every primitive data type of the *JSON* format, but also randomize the generated values, according to preplanned constraints, allowing the final values to be decided in execution time and not *a priori*.

XML Schema

The *XML* format assumes an hierarchical structure, where *XML* elements are nested inside others, all the way up to the root element of the instance, similar to a *JSON* object. Each element can possess attributes, as well as a text value and/or other elements nested within it, acting akin to a node in an information tree.

An element with complex content corresponds to the nesting of one or more *XML* elements within itself and is the closest there is to a *JSON* object in this syntax. This also implies that the notion of array does not exist in this data format: to adapt this structure from *JSON* to *XML*, it is necessary to arrive at a compromise. The approach adopted by *DataGen* consists in converting each element of the array in a key/value pair, where the key is a label of the respective value's index in the original structure.

Therefore, it is concluded that *XML Schema*'s primitive data types are all elementary (non-compound, unlike *JSON* objects and arrays). There is, however, a wide variety of these, in contrast to the five elementary types of values distinguished by *JSON* (figure 3), as can be observed in the following image:

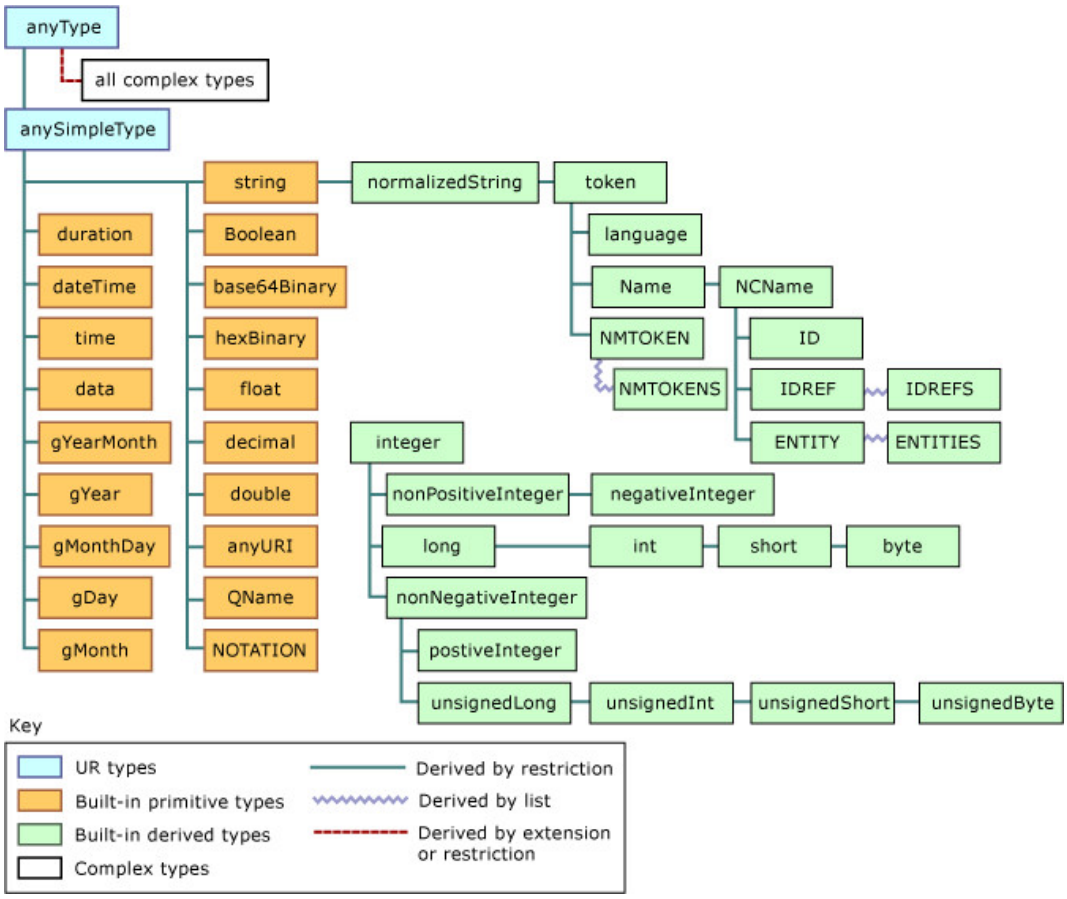


Figure 6: Primitive data types of the *XML* format.

After carefully analysing the chart presented above, it is possible to deduce that all of these types can be divided into several categories:

- **boolean**: Boolean;
- **integer**: integer and all of its derived types;
- **number**: decimal, double, and float (integers are also contained in this category);
- **string**: string, normalizedString, and all of its derived types, QName, and NOTATION (these last two have specific meanings in the context of the schema, but are strings regardless);
- **binary**: base64Binary and hexBinary;
- **temporal**: duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, and gMonth;
- ***URI***: anyURI.

Taking into account that binary, temporal, and *URI* types are also represented by formatted strings, despite their content having a specific meaning associated with the type in question, it can be concluded that the *XML* primitive types are basically a deep branching of the elementary *JSON* types, the only difference being that there does not exist any notion of null. Thus, it is possible to state the *DataGen* possesses, theoretically, the capacity to generate any of these types, as was explained in 2.2.2.

However, it is necessary to ensure that it also has the means to restrict the value to be generated to the lexical space of each primitive type, in order to obey the hierarchy presented in figure 6: for example, any token is a valid normalizedString, but the opposite is not true. Moreover, it is once again important to be able to induce controlled randomness in the generation procedure of any of these types. By analysing all of the types case by case, it can be infered that the following types have direct translation to *DataGen*'s *DSL*, through its interpolation functions:

- **numeric types** - functions integer and float, defining ranges of values correspondent to each type via their arguments. For example, the type byte translates to integer(−128, 127) and unsignedShort to integer(0, 65535);
- ***Boolean*** - boolean();
- ***date*** - function date, specifying the date format established by the type's lexical space, YYYY-MM-DD - for example, date("01−06−1950", "20−12−2010", "YYYY−MM−DD");
- ***dateTime*** - function date without the date formatting argument, which produces a date in raw format, e.g. date("01−06−1950", "20−12−2010");
- ***time*** - function time with arguments that constrain the value to the type's lexical space, for example time("hh:mm:ss", 24, false, "12:00:00", "20:30:00");
- ***gYear*** - function formattedInteger, whose third argument guarantees that the produced integer always has the number of digits indicated (in this case, 4), e.g. formattedInteger(1700, 2030, 4, "");
- ***gMonth*** - interpolation of two hyphens with the function formattedInteger, to generate values like "–06": '−−formattedInteger(1, 12, 2, "")';
- ***gDay*** - interpolation of three hyphens with the function formattedInteger, to generate values like "—15", e.g. '−−formattedInteger(1, 31, 2, "")';
- ***language*** - function language, which chooses from a list of abbreviations of the most frequently used languages.

All of the remaining primitive types are strings defined by very specific lexical spaces. For these cases, another important functionality of *DataGen* will be used: ***JavaScript* functions** - these allow the user to instantiate the value of a model property as the result of a function, where any intended code can be written, which is later resolved during execution, originating the final value. This way, it is possible to translate the rules of each type's lexical space to a set of operations and logical conditions, creating code snippets that generate random values of the remaining *XML* types.

Concluding, it is also possible to generate any primitive data type of *XML* Schema with *DataGen*. As such, the application fulfills requirement 3.

2.2.3 *Generation of Recursive Structures*

DataGen From Schemas must not allow infinite recursion in structures, since it needs a stop condition for the generation procedure, otherwise it would end up stuck in an infinite loop. As such, it will be necessary for the program to implement a restriction over the maximum depth of recursive specifications in schemas. For the sake of convenience, a setting for the minimum level of recursion produced in such datasets will also be provided. These choices will be made available to the users in the application's front end, in a settings menu, where they can adjust them to their liking, according to their particular use cases, managing the balance between the intended structure depth and the complexity/generation time of the dataset.

This way, the maximum depth of a dataset will be decided *a priori*. Taking into account that the *JSON*-like syntax of *DataGen*'s *DSL* makes it possible to specify any intended level of nesting of data structures, the ability to generate recursive structures is thus assured, under a set of reasonable and essential conditions (requirement 4).

2.2.4 *Fuzzy Generation*

The *Domain Specific Language* of *DataGen* provides several tools of fuzzy generation that are able to restrict the very existence of model properties in the produced instance based on either logical or probabilistic conditions (Santos et al., 2021). These functionalities are necessary to deal with some peculiar mechanisms of schemas, some examples of which will be presented below, pertinent to *XML* schemas:

Choice Elements

An element of asset type choice allows one and only one of its nested elements to appear in the final instance, i.e. it represents a set of mutually exclusive elements. *DataGen*'s *DSL* provides the directive or, which has exactly the same functionality, randomly choosing one

of its specified properties when generating the dataset. As such, there is a direct translation between these two features, which allows for a random element of the set to be chosen in runtime, thus making it possible to produce heterogeneous results from the same *DSL* model.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="employmentStatus">
    <xs:complexType>
      <xs:choice>
        <xs:element name="worker" type="xs:string"/>
        <xs:element name="employee" type="xs:string"/>
        <xs:element name="self_employed" type="xs:string"/>
        <xs:element name="director" type="xs:string"/>
        <xs:element name="office_holder" type="xs:string"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  employmentStatus: {
    or() {
      worker: '{{stringOfSize(5,50)}}',
      employee: '{{stringOfSize(5,50)}}',
      self_employed: '{{stringOfSize(5,50)}}',
      director: '{{stringOfSize(5,50)}}',
      office_holder: '{{stringOfSize(5,50)}}'
    }
  }
}
```

Figure 7: Translation of a choice element to *DataGen*'s *DSL*.

All Elements

An element of asset type all defines a set of unordered *XML* elements, i.e. they can appear in any order in the final instance. This will be implemented using the *DSL*'s tool at_least, which allows for a random subset of properties to be produced, by specifying the original set of properties and the minimum number of elements to be selected for the subset. An essential detail of this functionality is that the properties are chosen randomly and added to the final dataset by the order in which they are selected. This can be taken advantage of by translating an all element from a schema to an al_least directive that selects every element of the set, in the *DSL* model, effectively shuffling them.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="zooAnimals">
    <xs:complexType>
      <xs:all>
        <xs:element name="elephant" type="xs:string"/>
        <xs:element name="bear" type="xs:string"/>
        <xs:element name="giraffe" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  zooAnimals: {
    at_least(3) {
      elephant: '{{stringOfSize(5,50)}}',
      bear: '{{stringOfSize(5,50)}}',
      giraffe: '{{stringOfSize(5,50)}}'
    }
  }
}
```

Figure 8: Translation of an all element to *DataGen*'s *DSL*.

2.3 RELATED WORK

In this section, a thorough analysis of several real cases studies with the same finality as *DataGen From Schemas* will be carried out, in order to understand how versatile and complete the existing solutions are, at present. The focus will be on generating semantically cohesive data, since both specification languages have a high complexity and taking into account that *DataGen* already guarantees an efficient and scalable data generation procedure. A first subsection will highlight some of the most popular tools for generating *JSON* documents from *JSON* schemas, followed up by another that explores their *XML* equivalents.

2.3.1 *JSON Generators*

JSON Schema is a much newer concept than *XML* Schema, which was originally published in 2001. As such, it is natural that the data generation tools from *JSON* schemas that exist presently are not only fewer in number, but also of lower quality overall.

It is important to distinguish between ***JSON* Schema validators** and **data generators from *JSON* Schema**. There are several intuitive and comprehensive programs online for validating schemas, namely Hyperjump JSV, json-everything, jsonschema.dev, and JSON Schema Lint. However, these applications are not capable of generating data from schemas, they serve a different purpose - validating *JSON* documents against schemas (and possibly vice versa). Therefore, these tools have great practical value for the development of this dissertation, making it possible to test numerous instances and application cases of schemas, in order to better understand their syntax and purpose, as well as to certify that *DataGen From Schemas* is creating correct and compliant data, but they do not have the same function as the intended application. For this reason, they will not be addressed in this dissertation.

On the same note, *DataGen From Schemas* should also not be confused with **schema generators from *JSON***, for example JSON Schema Tool and JSON Formatter, as these programs implement the opposite workflow of what is intended, generating schemas from instances instead.

Finally, it is relevant to mention that there do exist some software packages for generating fake *JSON* data from schemas, such as json-schema-generator and json-schema-faker. However, these packages only implement a very old and outdated version of *JSON* Schema, draft-04. Since then, there have been published several further versions of the specification which are not backward compatible - draft-04 has a vastly smaller range of features compared to the current draft 2020-12 and some of its keywords now have entirely different semantics, such as the keyword items. As such, these data generation packages are not viable at present and in the context of *DataGen From Schemas*, since the application is projected to have an up-to-date and complete *JSON* Schema parser.

Moving on to the examination of some existing data generators, two different applications were selected to be analysed in this subsection:

JSON Schema Validator and Generator - ExtendsClass

This program is extremely basic and has little to no utility in the paradigm of data generation. It demonstrates an enormous lack of functionalities, given that it does not implement generation mechanisms for a great percentage of the *JSON* Schema syntax, and the functionalities that it does possess have a very limited and reductive implementation:

1. the application is divided into two major components: a *JSON* generator from *JSON* Schema (and vice versa) and a *JSON* validator, that checks against the inserted schema. Ironically, these two tools that should complement each other in the program's workflow frequently contradict themselves, when the validator determines that the data created by the generator is not a valid instance of the very same schema from which it was infered, which immediately highlights the poor nature of this generation program;

2. the generation of values is not random. All *JSON* content that the application is capable of generating boils down to an extremely basic set of default values - number: `0.0`, integer: `0`, string: `""`, boolean: `true`, object: `{}`, and array: `[]`. The minimal effort behind this generation procedure implies numerous serious errors in the resulting instances, as will be explained below;

3. it is not capable of generating any content when the `type` keyword is specified using array notation, to restrict the instance to one or more primitive types, for example `{ "type": ["number"] }` or `{ "type": ["number", "string"] }`;

4. the generation of enumerations is deterministic, since the program always chooses the last value of the enumeration. Furthermore, it is also faulty, given that it mistranscribes the value in consideration to the *JSON* document most of the time: if it's a string, the program does not wrap it in quotation marks and generates an invalid value; in case of an array, it does not wrap its elements in square brackets; if it's an object, it produces `[object Object]`, which means the program tried to write a *JavaScript* object directly to a string without first converting it with the function `stringify`;

5. when generating strings:

   - it ignores length keywords: `minLength` and `maxLength`. Since any string value generated defaults to an empty string, any set of restrictions for a string with length greater than zero implies that the created value is not semantically compliant;

- it is not capable of generating the syntax's embedded string formats: date–time, time, email, hostname, etc.

6. when generating numbers, it does not respect any of the numeric type keywords existent in *JSON* Schema, namely multipleOf (defines a number of which the instance's value must be a multiple of) and range keywords: minimum, maximum, exclusiveMinimum, and exclusiveMaximum;

7. when generating arrays:
   - it is not capable of producing valid data from array schemas with tuple validation, whose purpose is to specify a collection of items where each one has a different schema and its ordinal index is meaningful. Regardless of how many items are semantically defined in the schema, the program always produces a simple empty array, which is not even valid in this context;
   - it ignores the keyword additionalItems, which indicates if the structure is allowed to have more items besides those manually specified or not. This keyword is effectively futile from this generator's perspective;
   - likewise, it ignores all the remaining array type keywords of *JSON* Schema.

8. when generating objects:
   - it never generates any additional property, even when the keyword additionalProperties is set to true, which makes this keyword useless in this program;
   - even if a subset of object properties is declared mandatory with the keyword required, the solution always produces either all of the object's properties or only those that are required (the option is given to the user via a checkbox in the interface). It does not determine the existence of optional properties randomly (for example, through probabilities), which makes the result very predictable and uninteresting;
   - it ignores the keyword propertyNames, which allows the user to determine a text pattern to which all of the object's keys must conform, possibly generating properties with keys that are not compliant with the set pattern;
   - it ignores object size keywords minProperties and maxProperties.

Taking into account that the generation of the primitive data types in *ExtendsClass*'s application is extremely limited and rather defective, as was discussed above, further testing of more complex functionalities is not justified, since they do not possess a sturdy foundation in order to work properly. This tool is very rudimentary and its implementation of the *JSON* Schema syntax leaves a lot to be desired, so it exhibits very little utility for generating data from schemas.

JSON Schema to JSON Converter - Liquid Studio

This converter is an application provided by *Liquid Studio*, which also has some operating tools with *JSON* support, even though its main focus is on *XML*. Although there is an integrated development environment that centralizes this entity's generation tools, as will be mentioned again in 2.3.2, this *JSON* Schema to *JSON* program has not yet been built into the software, so its only available version currently is a free online program. As such, this was the version that was used for testing purposes, the results of which are presented below:

1. it introduces randomness in the generation of values of the elementary primitive types: numeric types, booleans and strings. Objects and arrays produced are always empty, unless further semantical restrictions are imposed on their content, in the schema. However, when generating an instance from a schema, the application caches the result and reuses it in subsequent generation attempts from the same model. For one to obtain different outcomes, it is necessary to refresh the page and input the same schema again for every new instance intended, which makes this operation more time consuming and worsens the user's experience, retracting value from the program's ability to create varied instances from the same initial state;

2. unlike the previous program, it is able to correctly process the keyword type when its value is specified in array notation, even choosing the data type to be generated randomly when the schema restricts the *JSON* document to more than one type;

3. the generation of enumerations is also defective: sometimes it works correctly and randomly selects one of the values of the keyword enum, at others it generates a new arbitrary value that is not specified anywhere in the schema;

4. when generating strings, it abides by the length keywords minLength and maxLength, creating strings with lengths within the established limits. Even so, it also does not support the generation of *JSON* Schema's built-in string formats (keyword format), which is a major drawback for the program's application cases and for the ease of instantiation of formatted strings;

5. when generating numbers, it does not show any improvements over the *ExtendsClass* software - the range keywords and multipleOf are also ignored in this case, creating values that are not compliant with the restrictions imposed;

6. the generation of arrays has little to add compared to the previous application's:
    - it obeys the keywords minItems and maxItems, generating structures with lengths within the established range. However, this funcionality is not very flexible, since the generated array values are all random, including their data types. The program

is not able to combine the length constraints with other semantic constraints relating to the array's content, e.g. the keywords contains or prefixItems;

- it seems to respect the keyword uniqueItems, since all generated items are random and, in all tests performed, no two items of the same array ever coincided;
- nonetheless, this solution is also missing a big percentage of *JSON* Schema array functionalities, namely tuple validation, random generation of additional items (when additionalItems is set to true), and the inclusion keywords contains, minContains, and maxContains, which are simply ignored in the current version.

7. when generating objects:
   - it obeys the specification of properties' contents according to name patterns (patternProperties), which the previous application did not. For example, if a property's value is defined as a string and there is also a pattern rule that covers the name of the property in consideration and maps its value to a numeric type, the generator recognizes the inconsistency and throws an error;
   - it also ignores additionalProperties and never generates any additional properties, which makes this keyword redundant in this context of data generation;
   - it always generates all properties specified in the schema, whether or not they are required - in this aspect, it even has a disadvantage compared to the previous program, which allowed the user to predefine if the final instance should contain all optional properties or none of them;
   - it ignores the keyword propertyNames, possibly generating properties with names that do not comply with the indicated standard, as well as the object size keywords minProperties and maxProperties. It does not correct these shortcomings that also existed in the previously analyzed generator.

It is possible to conclude that, despite showing some improvements over *ExtendsClass*'s application, which in themselves already make this tool more interesting and versatile, it still lacks in many important areas of the generation of *JSON* Schema's primitive types. These functionalities can affect not only each other, but also other, more complex tools of the syntax built upon them, for example schema composition and conditional application of subschemas, which makes their incorrect or non-existent implementation a serious flaw of the software.

As such, it is intended to address the issues exposed in this section on *JSON* generators in *DataGen From Schemas*, in order to achieve a more consistent, powerful, and flexible product.

2.3.2 *XML Generators*

This subsection aims to explore the advantages and disadvantages of two different applications whose purpose is to generate *XML* data from an input *XSD* model, arguably the most popular options available presently.

XSD2XML: Online XSD to XML generator

This tool proved to be quite disappointing, exhibiting a lack of numerous features that are important to the data generation process and a weak and rather limited implementation of the existing functionalities. Next will be exposed the downsides to this free web application, ascertained through its extensive testing:

1. it performs syntatic validation on the schemas input by the user, to check if the specification is well-constructed, and also some degree of semantic validation, however it is not very flexible - for example, in the case of a recursive schema, it throws an error to warn the user that the schema is invalid, although it is not. A possible solution for this issue would be to have an imbedded recursion limit in its generation algorithm;

2. it does not recognize all the base *XSD* data types: it does not support any type derived from normalizedString (token, language, Name, ID, ...). Some of these types are rarely used, but the absence of types such as ID and IDREF, for example, is very significant, since it automatically excludes the possibility of creating datasets with identification and reference of elements;

3. the generation of values is not random. Each data type has predefined values, according to which the *XML* content is produced: *string* -> 'str1234', *int* -> 123, etc;

4. limited support for restrictions of types - this tool is not reliable when it comes to generating user-constrained types, since the produced content rarely respects the restrictions specified in the schema;

5. the generated lists are invalid - as was approved and published by the *World Wide Web Consortium* (*W3C*), any type of *XML* list must have its items separated by whitespace. However, the values of the lists created by this program are all adjacent, as well as predefined, and there is no delimiter separating them;

6. it is not able to generate unions;

7. it does not randomize the order of elements of asset type all's contents, simply generating the elements in the order in which they appear in the schema;

8. it ignores occurrence attributes completely, generating all elements a single time. This is especially crucial, for example, when it comes to elements of asset type sequence - it generates sequences elementarily, with a single occurence of each element, even if the minimum number of occurrences specified for the entire sequence is greater than one;

9. it is not capable of generating choices - even though it apparently lets the user decide between generating only the first element of a choice or all of them, none of the tests carried out were able to accomplish either of these options: the element with the choice always came up empty and none of its nested elements were ever generated;

10. when the schema has semi-structured content, it does not generate any enveloping text, just the mixed elements themselves. Although this is technically compliant with the schema and a valid iteration of mixed content, it is very reductive and not representative of the possible results. Furthermore, and circling back to item 4, if the content of the mixed element's type is emptiable and the type is restricted so that it has no child elements and has non-empty surrounding text, the generated data is invalid;

11. it completely ignores the attribute nillable, even when it is set to true - it never generates the respective element with an attribute nil, only its regular content;

12. does not support cross-referencing between schemas (with the keyword include), since there is no functionality for file upload or any sort of workaround to enable such a feature;

13. it allows the user to predefine customized values for each of the supported data types. Still, even though this is a good addition to the program, it is redundant and its necessity arises solely from the application's inability to correctly process type restrictions, since this exact funcionality can be achieved directly in the schema, by restricting a simple type to a set of values with an enumeration.

After verifying all these flaws in the application, it was considered unnecessary to proceed with further testing, since little value can be extracted from it. *XSD2XML* is an extremely basic and simplistic data generation tool, capable of rigorously complying with only the most elementary schemas. It does not implement extensive and relevant parts of the *XSD* syntax, such as restrictions of simple types, and it does not introduce any kind of randomness whatsoever in the data generation prodecure, given that the result of every intermediate operation is predefined and limited. Furthermore, the solution is also not capable of generating instances with large quantities of data, always seemingly creating as little data as necessary/possible, which results in uninteresting datasets of little use.

XSD to XML Converter - Liquid Studio

As mentioned in 2.3.1, *Liquid Studio* provides an advanced toolkit for development in *XML* and *JSON*, complete with resources for mapping and transformation of data. Among its many tools, this toolkit possesses a *XML* data generator from schemas (*XSD*). Although this software is commercialized and its purchase is required in order to freely enjoy it, it is possible to sample the full product during a 15-day free trial period. This was the method utilized to gain access to the application for testing purposes, as a means to ensure that the results obtained corresponded to the best capacity of the software. There is a free online version of this program available at `http://www.liquid-technologies.com/online-xsd-to-xml-converter`, however the tests carried out on both alternatives proved that the toolkit version is superior and solves some issues of its alternative. For this reason, this document will disclose the conclusions drawn from the purchasable version of the product.

This generator proved to be far superior to *XSD2XML*, correcting many of its shortcomings and proving to be a more robust and versatile tool:

- it implements the whole *XML* Schema syntax, which means it supports every embedded simple data type (including types derived from normalizedString), as well as every kind of *XSD* elements, namely choices and unions, which the previous program was incapable of processing;
- it introduces a certain degree of randomness in the generation of data - whenever it produces a new *XML* document from a schema:
  - it creates values of certain simple types randomly, at runtime, e.g. numeric types;
  - whenever an element has an arbitrary number of occurrences (for example, minOccurs="3"maxOccurs="10"), the number of times it manifests in the final instance is determined randomly, within the imposed limits, somewhere along the pipeline.
- it incorporates an effective system of schema inclusion: whenever a schema with an include is input, the software automatically searches for the linked file in the same directory and, if it exists, imports it into the text editor environment and takes into account its contents when generating data from the original schema; if it does not find the linked file, it warns the user that it is not possible to find the necessary file in the working directory.

Furthermore, *Liquid Studio*'s converter possesses yet another very useful functionality that was not mentioned in the analysis of *XSD2XML*, due to its total absence in that software: it prompts the user with a set of customizable options, which allows to configure important settings such as the upper and lower limits of recursion and the maximum size of the file to

be generated, at which the program will stop generating optional elements. This menu gives the user more freedom and provides better control over the generated dataset.

Nevertheless, this tool still has some relevant flaws that prevent it from being as versatile and powerful as it could be, that will be addressed in the implementation of *DataGen From Schemas*. These shortcomings are listed below:

1. like *XSD2XML*, it performs syntatic validation of schemas, in order to check that their contents are well-formulated, but only partial semantic validation, not reporting any errors in certain invalid models - for example, a schema in which a new list type is declared, whose base type is also a list type, which is forbidden;

2. not all generated content is random - string and its derived types all have a default value ("string"), except for the element identification data types (ID, IDREF, IDREFS), which does not contribute for the variety and flexibility of the produced instances;

3. the values generated for the referencing data types IDREF(S) are invalid - for *XML* documents of a schema that has ID and IDREF elements to be valid, the contents of all IDREF elements must match the content of some ID element in the file, thus referencing an actual existing identifier. This tool generates ID values according to the following convention: "AAAA", "AAAB", "AAAC", etc; meanwhile, IDREF values are not the same, following a different pattern: "ID001", "ID002", "ID003", etc. As such, these identifiers and their supposed references never match, so the instances produced are semantically invalid;

4. the generated lists are invalid, the issue being exactly the same as with the previously analyzed application: there is no delimiter separating the list items, so each list can pass, at most, as a single-item list. However, a (minimum) length constraint greater than one is enough for the results to not be compliant with the schema;

5. related to the previous point, it was verified that the generation of lists is inconsistent - with certain schemas (valid, of course), the program generates empty *XML* elements and does not produce any kind of list;

6. it always generates the contents of an element all in the order in which they are specified in the model, instead of randomizing it - although compliant with the schema, it is an uninteresting approach to generating these elements, since their whole purpose is precisely to allow their nested elements to appear in any possible order;

7. it is incapable of generating complete semi-structured content: it produces only the nested *XML* elements inside the mixed element, without any enveloping text (same issue that was described in *XSD2XML*'s item 4);

8. it ignores the attribute nillable in elements - even when set to true, the solution always generates the element with its regular content and never with the attribute nil instead.

In conclusion, *Liquid Studio*'s *XSD to XML Converter* is a very advanced tool, capable of handling intricate use cases, however it is not without some unfortunate shortcomings that limit the usability of the software and compromise the reliability of the generated data. Nonetheless, it is a good application to study and serve as inpiration and guidestone for *DataGen From Schemas*, and it is possible to elicit a considerable ammount of functional requirements and quality markers for this type of program from it.

The conclusions drawn from the analysis of this tool, as well as the previous ones addressed in the present chapter, will be taken into account during the development of *DataGen From Schemas*, aiming to correct all the flaws observed in the semantics spectrum, thus creating a more robust and reliable application and implementing the most advantageous and useful features observed across all programs, for example the menu of customizable settings over the intended dataset.

Now that the application's purpose and general method of operation has been contextualized and an elaborate requirements elicitation has been carried out through the study of several relevant alternatives created for the same needs and use cases, the following chapter will propose a thorough approach to the development of *DataGen From Schemas*, detailing the application's expected workflow, architecture and design choices, as well as the main set of operations envisioned for the software.

3

PROPOSED APPROACH

DataGen From Schemas will be very similar to its prior version, *DataGen*, in terms of structure and workflow, since the core of the user interaction is exactly the same for both applications. To expand on this, the intended workflow for this new version is depicted in the following image and will be explained below:

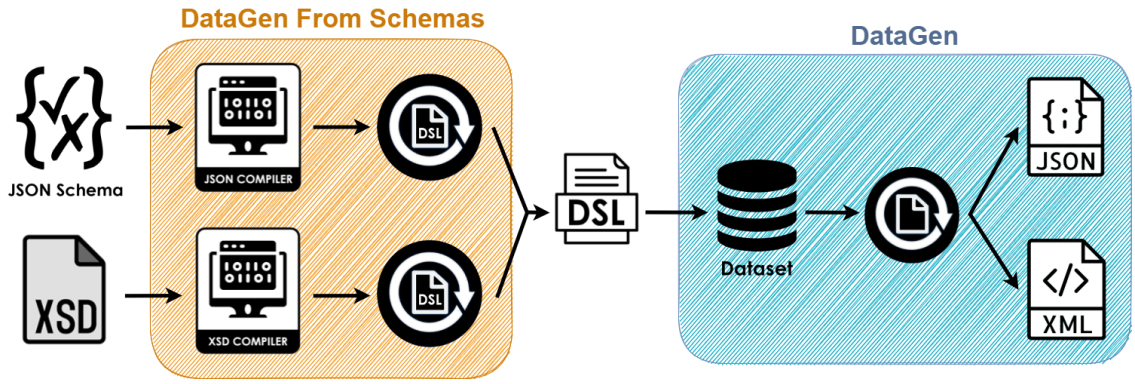


Figure 9: Workflow of *DataGen From Schemas*.

The program will accept the user's input of a *JSON* or *XML* schema, which will then be parsed by a compiler. The parser will generate an intermediate data structure with the relevant information and a converter program will then translate it to a *DSL* model. Afterwards, *DataGen* will take care of the remaining workload, parsing the model and generating a dataset, finally converting it to either *JSON* or *XML* format, according to the user's preference.

As with *DataGen*, it is intended to avoid building a single monolithic server, which would have implications in terms of performance and possible bottlenecks in the program. *DataGen From Schemas* will adopt a compartmentalized architecture, which prioritizes the scalability and availability of the application, separating its front end and back end on distinct servers. This way, the failure of an individual component does not compromise the functionality of the entire application, which also makes it easier and faster to perform maintenance routines.

This fragmented architecture makes it possible to provide a more pleasant and consistent user experience and also provides room for the eventual enhancement of any of the individual servers, by assigning them more computation resources and fault tolerance mechanisms, such as load balancing and redundancy, should the need arise to scale the application in order to deal with traffic and/or availability issues.

3.1 ARCHITECTURE

The proposed architecture for *DataGen From Schemas* is reflected in the following diagram and will be explained in detail throughout this chapter, seeking to justify the chosen technologies and the designed structure.

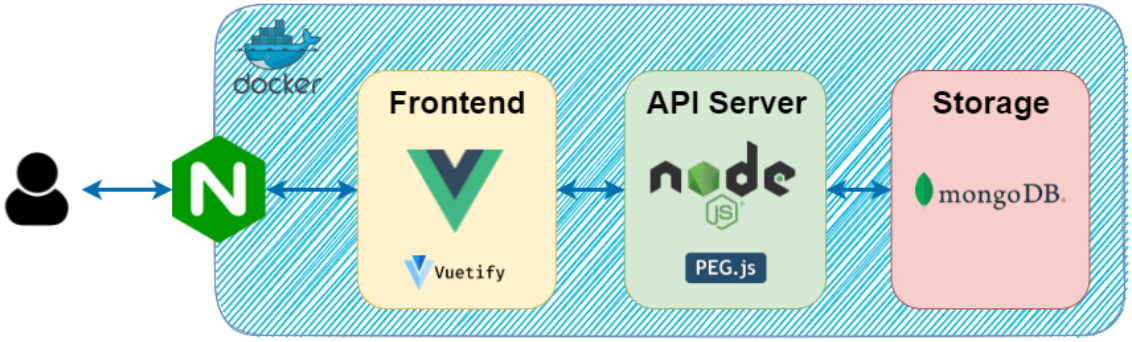


Figure 10: Proposed architecture for *DataGen From Schemas*.

Since *DataGen From Schemas* is projected to be a complementary application to *DataGen* with the goal of expanding the original program's application cases, in order to enable the generation of data from schemas, it is intended to keep some of the functionalities that already exist in *DataGen*, namely the user authentication, the option to save *DSL* models and their availability on the platform. It does not make sense to also save schemas in the platform, associated with the respective user's account, since these are files of standardized and globally recognized formats (*JSON* and *XML*), unlike the *Domain Specific Language*, which is only recognized in the context of the application.

Taking into account that all the procedures behind storing and managing this data are already implemented in the original application, a separate new implementation in *DataGen From Schemas* is not justified, since it would just be a copy of most of the existing work. On the contrary, the hypothesis of reusing *DataGen*'s back end and database proves to be much more interesting and useful, given that it allows this new version to maintain the users' records and *DSL* models from the original application, ensuring the cross-platform sharing of all persistent data between the two applications, as well as avoiding a lot of repeated

and unnecessary work. This approach only brings advantages from the users' point of view as well, since it will allow them to use the same account in both applications and to edit models from both *DataGen* and *DataGen From Schemas*.

The deployment of *DataGen* was performed via *Docker*, which allows the instantiation of the project through containers - structures independent from the operating system -, placing each component of the architecture in a different one. *Docker* encapsulates the application and is accessed by an instance of a *NGINX* server, which acts as a proxy of requests to the program's front end and back end. The database is hidden from *NGINX* itself and is completely invisible/inaccessible to any entity outside the *Docker* instance.

This structuring implies a further advantage for the reusage of *DataGen*'s back end, since it facilitates the access to the original database, in order to use the same data. As such, the idea is to develop a new front end for *DataGen From Schemas* and insert it the original version's *Docker* instance, finally linking it to the application's back end in the *Docker Compose* (a tool that helps define and share multi-container applications), in order to implement persistence of data and allow data sharing between both versions, while keeping the data layer-secure and hidden from outside entities.

3.1.1 *Server-sided Data Generation*

The processing of schemas will be performed by compilers and it is necessary to develop distinct compilers for *JSON* Schema and *XML* Schema, due to their fundamentally different syntaxes, as was explained in section 2.1. These compilers will be based on *PEG.js* grammars, since this was the technology used to develop *DataGen*'s original *DSL* compiler and proved to meet all the requirements of the program. Thanks to this, the author of this dissertation is also already quite familiar with the tool and knows its various features well, so there will not be the typical initial learning curve and it will be possible to proceed to the development phase more quickly.

When parsing the file, the compilers will produce an intermediate data structure with all the relevant data extracted from the schema. This structure will then be passed to a converter program written in *JavaScript* (again, there will be one for each format), for direct compatibility with *PEG.js* (that also incorporates this programming language) and *JSON* (which has built-in support and conversion to *JavaScript* objects), which will then translate it into a model of *DataGen*' *DSL*, that will later be processed by the base application to produce the final dataset. This workflow can be observed in diagram 9.

Even though *DataGen*'s published article (Santos et al., 2021) postulated a client-sided approach, where the computational burden of generating data would be placed entirely on the client's browser, this description is outdated and does not reflect the present structure of the application. Subsequently, the data generation procedure was moved to the back end of

the program, adopting a server-sided approach, in order to be able to expose *API* routes without having to clone the compiler based on the *PEG.js* grammar and place a copy on each of the application's servers.

Following the same logic, it will also make more sense to place the compilers and converters of *DataGen From Schemas* in the back end of the application, in order to avoid duplicating these files to allow the creation and exposure of new *API* routes, thus enabling the usage of this service without recourse to the interface and its eventual integration in third parties. On the one hand, these programs are independent from the rest of the application and perform their function individually, without needing to access any of the back end's private services, namely the database or local variables, which suggests a client-sided approach to the processing of schemas. However, the overall process of data generation from schemas will never fully be client-sided, due to *DataGen*'s implementation (generates the dataset in the back end), so this differentiation is not justified.

Furthermore, the computational load of processing the schemas and generating the *DSL* models is negligible, so the gains derived from performing these operations on the client's browser (in terms of freeing up resources on the back-end server, mainly regarding *CPU* and memory) are trivial. Placing the new compilers and converters in the back end shared with *DataGen* also allows to centralize the programs' functionalities, avoiding additional requests between servers and possible downtime, as would be the case if *DataGen From Schemas* was deployed in an entirely separate *Docker* instance and had to communicate with *DataGen*'s *API* via *HTTP* requests.

Therefore, it emerges as a natural conclusion that *DataGen From Schemas* should adopt a server-sided approach for processing the schemas, creating the *DSL* models and consequently generating the respective datasets.

3.2 Front end

The front-end server is the entry point to the application and is responsible for compiling and displaying the platform's interface to the users, managing any and all user interaction.

In general, *DataGen From Schemas* is intended to have an interface very close to that of *DataGen*. This application is, at its core, an extension of its predecessor and its workflow is analogous - input and processing of model, data generation, output of dataset -, differing only in the type of input (schemas vs. *DSL* models). As such, there is no need to "reinvent the wheel" - it is possible to take advantage of the basis already established by *DataGen*, also for the sake of visual cohesion between the two complementary tools.

For this reason and others that will be mentioned below, it is proposed to implement the interface in *Vue.js*, which was also the technology used in *DataGen*'s front end. This

framework utilizes a *Model-View-ViewModel* (*MVVM*) architecture that simplifies the event-driven paradigm used to process the interactions between the user and the interface:

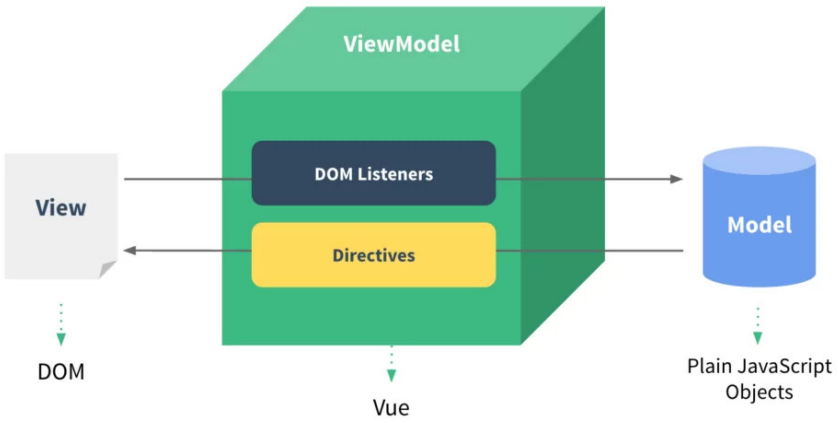


Figure 11: Representation of the *MVVM* architecture.

With this architecture, *Vue.js* effectively separates the user interface (*View*) from the program logic (*Model*), establishing a reactive bidirectional data connection between the two that circumvents the need to reload the page when it becomes outdated in relation to the data model. This makes it possible to translate changes in the model to the *Document Object Model* (*DOM*) in real time, updating only the corresponding interface elements, which results in a much more user-friendly experience, as well as more efficient performance than some of its alternatives, namely *React* and *Angular*.

In operational terms, the interface will provide the following features to the user:

- **Authentication** - implemented with the *JSON Web Token* (*JWT*) standard, which allows both servers to be stateless, bypassing the need to maintain sessions with user data. All *HTTP* requests to the back end that access sensitive data are signed with this token;
- **Generation and download of datasets from *JSON* or *XML* schemas** - the user will be provided with two distinct methods to input models: either manually, in the interface's text editor, or by uploading a schema file to the platform;
- **Error feedback** - the application will not only inform the users of any incompatibility of their models with the program's data generation procedure, but it will also semantically validate the schemas, reporting any errors it may encounter;
- **Saving *DSL* models** - in the original version, the users were able to save any *DSL* model (which would be associated to their accounts) and to later change their visibility on the platform and edit their contents. *DataGen From Schemas* will keep these features

on its machine-generated *DSL* models, providing the option to save these intermediate models, which can then be used in the original software.

3.3 BACK END

The back end server is responsible for implementing the application's business logic and managing its data persistence layer, which in this case stores the users' informations and the *DSL* models. As was discussed previously in section 3.1, *DataGen From Schemas* will use the same back end as its predecessor, expanding its functionalities with the parsing of schemas and consequent generation of *DSL* models, as well as *API* routes for these new use cases. In addition, the server will still be in charge of data generation and also the authentication of users and the management of their respective models.

The server is implemented in *Node.js*, which emerges as the ideal technology for this context - its event-driven, asynchronous and single-threaded nature grants it a distinct efficiency when handling requests with light computation load, as is the case. It also has good performance in terms of speed, a very active community (which translates into good documentation and help), good scalability, and an extensive package library.

As far as information persistence is concerned, there will be no need for modifications on *DataGen*'s database, which will also be accessed by the new application in order to keep the same user registry and to be able to manage all stored models together, from both platforms. The database is implemented with a *MongoDB* server instance, a technology that was originally chosen due to its *Not Only SQL* (*NoSQL*) document-oriented nature, highly compatible with *Node.js*, since both have support for *JSON* documents. It is endowed with remarkable scalability and performance, derived from the fact that it does not group data relationally, which allows fast, efficient, and conflict-free queries, since all documents are independent.

Persistent data is organized into three different collections:

- **users** - name, email, password (encrypted), and dates of registration and most recent access;
- **models** - *DSL* model, creator, visibility (public or private), title, description, and registration date;
- **blacklist** - stores users' *JWT* tokens when they log out, along with their expiration timestamp, so that they are automatically removed from the database as soon as they expire, preventing the users' sessions from remaining open forever.

Finally, the back end will also provide *Representational State Transfer* (*REST*) routes to enable the usage of the program without recourse to its graphical interface. In this way,

it will also be possible to incorporate *DataGen From Schemas*'s features into third party applications, through *HTTP* requests. The planned routes are the following, where the input schema will be sent in the request's body:

- **POST /api/json_schema/json** - produces a *JSON* dataset from a *JSON* schema;
- **POST /api/json_schema/xml** - produces a *XML* dataset from a *JSON* schema;
- **POST /api/xml_schema/json** - produces a *JSON* dataset from an *XML* schema;
- **POST /api/xml_schema/xml** - produces an *XML* dataset from an *XML* schema.

As can be seen above, *DataGen From Schemas* is intended to be able to generate data in both *JSON* and *XML*, from either of the two types of schemas. For this purpose, it will be necessary to establish some norms regarding the representation of *XML* information in *JSON*, in order to guarantee full data portability between the formats, due to some characteristics such as those mentioned in 2.2.1. As such, the following taxonomy is proposed for *JSON* instances generated from *XML* schemas:

- any occurrences of the characters . (dot) and - (hyphen) in property keys will be replaced with the character _ (underscore), since *JSON* does not allow dots or hyphens in property names, while *XML* does;
- properties with the same key at the same depth in a given structure will be differentiated using a counter and if a key is unique, it will not be numbered. For example, if an *XML* element possesses three nested elements with the key "address", in *JSON* these will become "address1", "address2" and "address3";
- attribute keys will be prefixed with a flag indicating that they are attributes: for example, an attribute "weight" in *XML* will become "attr_weight" in *JSON*;
- whenever an *XML* element has attributes and textual content, the content will be placed in a *JSON* property with the key value;
- whenever an element has mixed content, the text outside of the child elements will be placed in numbered properties text1, text2, etc. In case there is only one instance of semi-structured text, it will instead be formatted into a *JSON* property with the unnumbered key text.

With this architecture and technical requirements in mind, the project progressed to the development phase, looking to build the new components from the ground up and implement the logic and workflow that was described here, which will all be minutely covered in the next chapter.

4

DEVELOPMENT

The implementation of *DataGen From Schemas* was realized over several phases: firstly, developing the application's back end, whose core consists in the schemata parsers and respective translation programs to *DataGen*'s *DSL*. This was the first logical step, since the application needs these components to be able to carry out its function, with or without a graphical interface. Furthermore, setting up a duplicate build of the application's prior version running in **localhost** and developing these tools directly in its back end, where they would eventually be integrated, allowed for their immediate testing without the downtime needed to communicate with *DataGen*'s live version.

The next step was to implement the application's front end, designing and developing its graphical interface and setting up communication between its components and the back end. *DataGen From Schemas* also requires the notion of session, allowing users to register, log in and log out, just as its prior version, in order to enable them to save *DSL* models in their accounts. The interface was built as a *Single-Page Application* (*SPA*), since its array of features was compact and interconnected, which enabled the creation of an intuitive, user-friendly, and easy to navigate website with fast transitions and reduced load times.

Then followed the implementation of *API* routes in the back end, as well as the introduction of custom support for *DataGen*'s interpolation functions directly in the schemata grammars, in order to enable more detailed and specialized specification of dataset fields, as will be described in section 4.3. Afterwards, it was necessary to produce documentation on these two features in the application's front end, so that users have access to this information and can easily learn how to use them.

Finally, the application was thoroughly tested and debugged, using complex schemas and real-life examples to evaluate its performance, both with trivial schemas and under heavy stress, as well as its correct parsing of every particularity of the schemata languages.

4.1 JSON SCHEMA COMPONENT

In order to generate *DSL* models from *JSON* schemas, *DataGen From Schemas* needs two different components: firstly, a *PEG.js* grammar-based parser to analyze the schema and

extract all useful information; secondly, a converter program capable of building a *DSL* model from the intermediate structure.

The user may input one or more schemas into the program, since cross-schema referencing is supported. In this case, the user must indicate which of them is the primary schema from which the program must generate the dataset. The parser then analyzes all schemas sequentially and builds a separate intermediate structure for each of them, a procedure that will be explained in detail in this section.

4.1.1 *Grammar*

JSON Schema is a *JSON* vocabulary, i.e. it is written in *JSON* and operates under a strict set of rules, where specific keys have precise meanings and can be used to annotate or validate *JSON* documents. This common syntax makes it easy to interpret the schema and validate the instance.

Therefore, the grammar was built with a *JSON* grammar available in the *PEG.js* Github as its foundation. *JSON* Schema's specification is made available by drafts, which represent versions. Each time the vocabulary is majorly updated, a new draft is released, where new features can be found and existing ones altered. These drafts are not backward compatible, for example there are cases in which a certain key has entirely different semantics depending on the draft considered. As such, it seemed only logical to adapt the most recent draft to the application, which is, at the time of writing this dissertation, *JSON* Schema 2020-12.

As such, the aforementioned *JSON* grammar was modified into a dialect (a specific version) grammar of *JSON* Schema for this particular draft: the set of keywords and semantics that can be used to evaluate a schema was restricted to those made available in the draft and custom vocabularies defined by the user are not accepted.

Important Features

By restricting the keywords accepted by the grammar and their semantics, it was possible to implement a number of important features in this grammar, namely:

- **restriction of each keyword's value to its rigid lexical space** - for example, the keyword uniqueItems's value must be a boolean and nothing else, while the value of the keyword additionalProperties may be any subschema;
- **rigorous semantic validation of the schema** - in *JSON* Schema, it is possible to create invalid and contradictory schemas. This may range from something as simple as a number with a maximum of 20 and minimum of 50, to more contrived cases such as establishing that a schema must be both of type boolean and string, with the keyword allOf (example below). As such, a semantic validation procedure was created for this

grammar, that checks for erroneous combinations of values or other incongruities in the schema's logic;

Figure 12: Example of an invalid *JSON* schema.

- **error reports** - following the previous points, whenever the parser finds errors, whether syntatic or semantic, it halts the execution of the pipeline and reports them to the users, for them to correct and try again. It provides a detailed explanation for each error, including its position in the schema, what was expected and what was found instead.

Intermediate Structure

The other central focus of the grammar is building the intermediate data structure, to where it extracts all relevant information. Before addressing this topic, it is best to categorize and explain the different kinds of *JSON* Schema keywords. The reader is encouraged to follow this section of the paper along with the official *JSON* Schema documentation, as it has all the keywords listed and sorted in a relevant taxonomy. For this solution, the following categorization was taken into account:

- **type-specific keywords** - these are keywords that apply only to the data type in question. For example, numeric types have a way of specifying a numeric range that would not be applicable to other types;
- **generic keywords** - const, enum, and type. The latter defines the type(s) that the schema validates, the others may be or contain values of any of those types;
- **schema composition keywords** - the purpose of these is to combine together schemas and they correspond to well-known algebra concepts like AND, OR, XOR, and NOT;
- **keywords to apply subschemas conditionally** - based on logical conditions or the presence of certain properties in the final object;
- **structural keywords** - $id, $anchor, $ref, and $defs. These keywords do not reflect values of the instance explicitly, but are used to structure complex schemas, allowing the user to break them down into simpler, reusable subschemas, and to reference these

from anywhere, to avoid duplication and write schemas that are easier to read and maintain;

- **ignored keywords** - comments and annotation/media keywords (string-encoding of non-*JSON* data). The parser recognizes these keywords but willingly ignores them, since they have little to no use in a dataset generation context.

Since a schema's structure is very basic and consists only of key/value pairs, it is easy to store its information in memory, before reorganizing it into a more useful configuration for its following conversion to a *DSL* model. As such, when parsing a (sub)schema, the grammar firstly validates its content semantically and stores it in memory basically as is, in a *JavaScript* object, with only minor adjustments: e.g. the value of the keyword multipleOf is stored in an array, despite being a number, since the user may use schema composition to establish further multiplicity constraints, which will all be bundled together to restrict the instance jointly, thus it's useful to adapt the keyword's value to array format from the start, to facilitate parsing and standardize use cases.

The grammar then reorganizes the structure into a more efficient configuration. After thorough reflection, it was concluded that the best approach would be a **type-oriented structure**, where the keywords and respective values would be stored under the type of data they produce. There are multiple points in favor of this line of reasoning:

- each data type has a specific set of keywords that applies only to them (the aforementioned type-oriented keywords), so it is easy to separate most keywords by type;

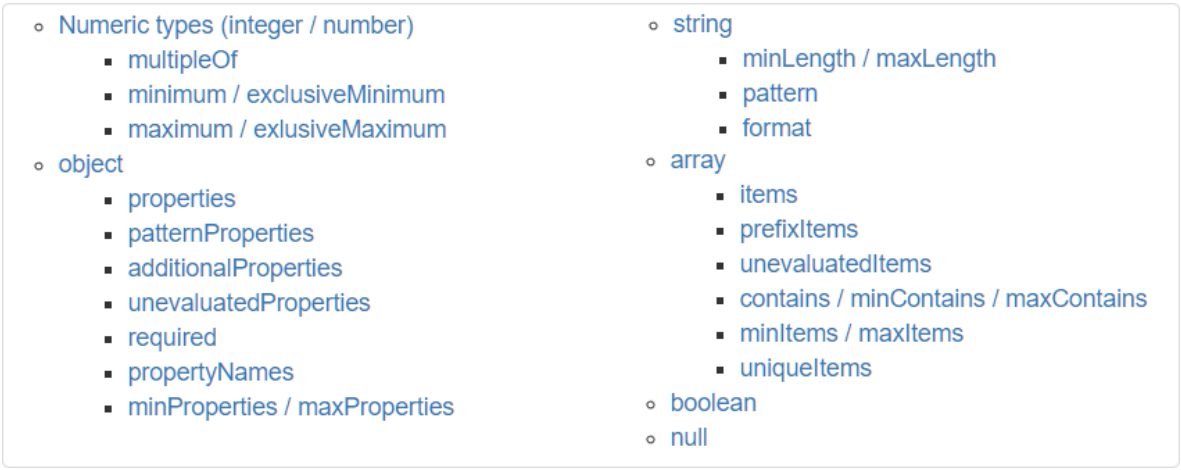


Figure 13: *JSON* Schema's type-specific keywords.

- a single *JSON* schema may validate against multiple data types - the same schema can have keywords respective to booleans, numbers, and strings, for example. As such, it is useful to know, at all times, exactly what data types are produceable from the schema:

```
{
  "type": [ "number", "string", "boolean" ],
  "multipleOf": 3,
  "minLength": 10,
  "minimum": 100,
  "maxLength": 20
}
```

```
{
  "boolean": {},
  "number": { "minimum": 100, "multipleOf": 3 },
  "string": { "minLength": 10, "maxLength": 20 }
}
```

Figure 14: Example of a multi-type *JSON* schema and its respective intermediate structure.

- following the previous point, a certain type may be established and then "disallowed" further into the schema, with a keyword of schema composition. As such, it would simply be removed from the intermediate structure, preventing its generation or further unnecessary parsing (example ahead in figure 18);
- this kind of organization makes it easy to update the structure for each new keyword parsed and facilitates the translation exercise executed later on. With this, the translation program will need only to choose a random type and parse the keywords associated with that type, ignoring all others. It is efficient and compact.

However, not all keywords are related strictly to a single type. Generic and schema composition keywords, as defined previously, plus if/then/else (that apply subschemas conditionally), may take values or subschemas of multiple types. In these cases, the grammar follows the ensuing method: firstly, it makes sure each of the keyword's values has a single type. For generic keywords, this is already the case, as its values are already the final product. However, the values of the other keywords mentioned are subschemas, which may be multi-type: if so, the grammar breaks down the subschema into smaller subschemas, one per each of its types.

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "string", "minLength": 10 },
    {
      "type": [ "number", "string" ],
      "multipleOf": 3,
      "maxLength": 20
    }
  ]
}
```

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 3 },
    { "type": "number", "multipleOf": 5 },
    { "type": "string", "minLength": 10 },
    { "type": "string", "maxLength": 20 }
  ]
}
```

Figure 15: Example of parsing done on multi-type values of a schema composition keyword.

Then, it separates the keyword's values by type and introduces an instance of said keyword in each of its generateable data types, in the intermediate structure, along with that type's respective values. An example of this is shown below:

```
{
  "number": {
    "oneOf": [ { "multipleOf": 3}, { "multipleOf": 5 } ]
  },
  "string": {
    "oneOf": [ { "minLength": 10 }, { "maxLength": 20 } ]
  }
}
```

Figure 16: Intermediate structure produced from the previous schema.

With this algorithm, the program is able to classify these keywords and, by extension, all *JSON* Schema keywords by type, which makes it possible to use the described structure to store all relevant data, in a way designed to facilitate and make more efficient the following program's translation routine.

In conclusion, the main objective of the grammar, and consequent parser, was to reproduce the *JSON* Schema syntax meticulously and collect data from any given schema to a well thought-out and efficient data structure, to set up the next phase of the process - the construction of the *DSL* model. Furthermore, it was also to make the solution as sturdy and fault-tolerant as possible, preventing it from trying to parse impossible schemas and crashing or producing unexpected behaviour, which in turn helps the user to better understand their schema and detect unwilling errors.

4.1.2 *Referencing*

JSON Schema references can vary a lot: there are absolute and relative references, depending on if they include the schema's base *Uniform Resource Identifier* (*URI*). It is not mandatory for a schema to have an id, which is its *URI*-reference, but without one, it is not possible to reference it in other schemas, although it can still have local references. Furthermore, a subschema may be referenced either by *JSON* pointer, which describes a slash-separated path to traverse the keys of the objects in the document, or by anchor, using the keyword $anchor to create a named anchor in the subschema to be referenced. The reader is invited to check out the official documentation on schema structuring, in order to gain a more in-depth understanding of schema identification and referencing, which is crucial for this component of the solution.

DataGen From Schemas supports all types of referencing defined in *JSON* Schema 2020-12 and standardizes that any schema's base *URI* must begin with `https://datagen.di.uminho.pt/json-schemas/` and be followed by the schema's name.

Besides the configuration exposed in 4.1.1, the intermediate data structure has two additional sections: one for storing the object pointers to all references found in the schema and another for separately storing all subschemas with their own declared identifier. This division is useful to resolve all references later on, after the parser has finished analyzing every schema submitted by the user. The program is unable to resolve references when it is reading the schemas, since they might be pointing to schemas or subschemas that have yet to be reached. For the sake of efficiency, instead of checking if that is the case whenever a new reference is found, the parser simply caches the references and subschemas, in a way that later on it will be able to quickly find each referenced subschema and substitute its content in the main body.

Thereby, it becomes possible to generate datasets from schemas with local and/or external references, as well as more intricate mechanisms, such as recursion and bundling.

RESOLVING REFERENCES

After parsing every schema, the program needs to resolve all existing references before moving on to the creation of the corresponding *DSL* model. In order to do this, it must determine an optimal order in which to approach the references, since some may be dependent on others, which means they can only be fully resolved after their dependencies are complete.

Naturally, the application starts by resolving the local references of every subschema first, since these point only to the own schema's structure, which means each one already has most, if not all the necessary information to handle such cases. When resolving a reference, the program copies the referenced structure to where it is being pointed from, eliminating the $ref property from the object and thus centralizing all the necessary information for a schema's translation to the *DSL* in its own structure. The referenced subschema is duplicated via a **shallow copy**, so that any existing nested references preserve the original object pointer when duplicated - this way, it is only necessary to resolve each of these nested references once, later on, and its new content will be reflected across all of its instances. Recursive references are parsed separately, by randomly determining, at this point, the level of recursion that the instance will possess, within reason, in order to avoid incurring in infinite loops.

After resolving every schema's local references, it moves on to their foreign references, with a newly-acquired assurance that the referenced content is as simplified as possible, at this point. The program builds a **dependency map** between all schemas and uses it as a

guide on which order it should go about the schemas, starting with those that reference isolated schemas, which in turn become "isolated" themselves, after all relevant content has been centralized in their intermediate structure, and repeating this process along the map until all references are finally resolved.

Also, it throws an error if it detects any infinite dependency loops between schemas, e.g.:

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/address",
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" },
    "inhabitants": {
      "type": "array",
      "items": { "$ref": "/schemas/person" }
    }
  },
  "required": [ "street_address", "city", "state", "inhabitants" ]
}
```

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/person",
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "address": { "$ref": "/schemas/address" }
  },
  "required": [ "first_name", "last_name", "address" ]
}
```

Figure 17: Example of *JSON* schemas with an infinite reference loop between them.

4.1.3 *Post-processing of the Intermediate Structure*

Once the intermediate structure of the main schema is finalized and all references resolved, it is then sent to the translator program to begin creating the correspondent *DSL* model. This component interprets the keywords present on the structure and generates a *DSL* string accordingly, taking into account how they influence each other. For the sake of brevity, the *JSON* Schema keywords will not be explained minutely in this dissertation, so it is strongly recommended to follow along this subsection with the official *JSON* Schema documentation, as it thoroughly details the function of every keyword, illustrated with meaningful and intuitive examples.

The intermediate structure is type-oriented, meaning that each value of the schema is described by a *JavaScript* object that maps each of its createable data types to their respective keywords, and values are organized in a hierarquical structure. To produce the model for the whole schema, the program recursively iterates this intermediate structure, generating its values' *DSL* strings from the leaves to the root and gradually merging them together.

The types present in the structure of each value already reflect the whole logic of its schema, since the parser relates the keywords and determines the generateable data types common to all of them, as described in subsection 4.1.1. Take the following example, illustrated below: even though the keyword type defines that the instance may be either a string or a number, the keyword allOf implies that only a number is valid, since all its

subschemas are only of that type. As such, the section of the intermediate data structure that describes this value will not have the string type:

Figure 18: Intermediate structure of an apparent multi-type schema.

The program then randomly selects one of the generateable types and moves on to translating its keywords. The first step is to resolve any existing keywords of schema composition or conditional application of subschemas - these keywords are not directly translated to the *DSL* string, but rather parsed and its contents added to the structure.

In *JSON* Schema, the aforementioned keywords are not used to extend or merge schemas, in the sense of object-oriented inheritance. Instead, instances must independently validate against each of the keywords. This is reasonable when validating instances against schemas, which is the purpose of *JSON* Schema. However, *DataGen From Schemas* reverses this workflow and looks to create instances from schemas, so the same logic does not apply. It is not possible to generate a different value for each of these keywords and ultimately merge the values together. This method could result in some values being valid against individual parts of the schema, but possibly not the whole of it.

As such, in the context of data generation, it is necessary to parse these "compound" keywords beforehand and extend the base schema with their content, obtaining a cohesive and coherent final schema with only type-specific keywords that incorporates these restraints. Since these "compound" keywords' values are or contain schemas, which may, in turn, have nested such keywords, the program recursively checks all subschemas for these keywords and resolves them, before using their content in the extension process, so that ultimately the base schema is extended only with type-specific keywords.

Schema Composition Keywords

There are four keywords belonging to this category: allOf, anyOf, and oneOf, that allow the user to define an array of subschemas, against all, one or more, or exclusively one of which the data must be valid, respectively; not, which declares that an instance must not be valid against the given subschema.

For the first three, a subset of their schema values is chosen: with allOf, all of its schemas are considered; for anyOf and oneOf, either an arbitrary number of its schemas or only one of them, respectively, are randomly selected. The base schema is then extended with these sequentially and the original keywords are erased from the structure.

As for not, the program must first "invert" this keyword's schema, in order to obtain a complementary/opposite schema, which ensures that no value that is valid against it also validates against the original schema. Then, the base schema is extended with this inverted schema and the keyword not is removed from the structure.

For this purpose, a schema inverter capable of generating complementary schemas was developed, which takes into account the meaning of each *JSON* Schema type-specific keyword. There is never a need to invert any other kind of keyword, since those are parsed recursively before the actual schema to which they belong, which guarantees that the schema to be inverted will only have type-specific keywords.

On the same note, the solution also incorporates a schema extender program capable of manually extending a base schema with each type-specific keyword. For each data type, the new keyword is compared to the already existing ones and incorporated in a reasonable way - the result may be different depending on whether the base schema already has the same keyword or not, for example. This solution must handle each individual *JSON* Schema keyword differently, as they all have different meanings.

These two components will be later covered in sections 4.1.5 and 4.1.6, after briefly going over every type-specific keyword in the explanation of the translation process to the *DSL*, for better understanding of how they relate with each other.

Keywords that Apply Subschemas Conditionally

These keywords are the reason for *JSON* Schema's **dynamic semantics**, i.e. their meaning can only be uncovered after the context has actually been instantiated, since these keywords establish conditions based on actual values of the instance.

Take the following example, where the schema will only know what pattern to validate the property postal_code with after checking the actual instance for the value of the property country, and never before. While with other keywords, the schema can determine *a priori* the structure of the instance, with these it is not possible to do so:

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "country": {
      "default": "United States of America",
      "enum": [ "United States of America", "Canada" ]
    }
  },
  "if": {
    "properties": { "country": { "const": "United States of America" } }
  },
  "then": {
    "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
  },
  "else": {
    "properties": { "postal_code": { "pattern": "[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" } }
  }
}
```

Figure 19: *JSON* schema with dynamic semantics.

Once again, this approach is not reasonable for a solution such as *DataGen From Schemas*, since it is not viable to generate an intermediate value from the rest of the schema and only then exert these keywords, which may imply large changes to the remaining schema - at least with if, then, and else. Since the keywords dependentRequired and dependentSchemas are specific to the object data type, it is possible to incorporate them into the translation procedure of object schemas, as will be detailed ahead in subsection 4.1.4.

The solution found for the keywords if, then, and else was to determine their outcome probabilistically - unless the if schema is explicitly true or false, in which case it is deterministic whether the instance should be validated with either then or else, respectively. Otherwise, the program determines the veracity of the condition based on a probability, customizable by the user (by default, a 50/50% chance) - if true, the base schema is extended with the if and then schemas, otherwise it is extended with a complementary schema of if (produced with the aforementioned schema inverter) and the schema of else. This way, it is possible to produce a coherent, simpler schema that incorporates the logic of these conditional keywords and to generate the final instance in a single iteration of the structure.

4.1.4 *DSL Model Creation*

Finally, once the intermediate structure of the selected data type is finalized and possesses only type-oriented keywords and/or const and enum, the program is able to generate a *DSL*

string that translates the logic of the original schema. This subsection will carefully detail the method behind the translation of each type of value.

DataGen's *DSL* was desgined for the specification of datasets and, as such, its models must obey a key/value notation, for the clear structuring of the intended instances. Therefore, this tool is only able to generate objects, which are later translated to whichever output format the user wants, and not any of the other native *JSON* data types, namely numbers, strings or arrays. To circumvent this, whenever the root schema introduced is not of type object, the program creates a *DSL* model with a single property, with the value being the schema's translation to the *DSL* and the key being DFJS_NOT_OBJECT (*DFJS* stands for *DataGen From JSON Schemas*). After *DataGen* generates the final dataset, its translator programs to either *JSON* or *XML* detect this key and get rid of the object structure, returning only its value, as intended.

```
{
  "type": "number",
  "minimum": 0,
  "exclusiveMaximum": 100,
  "multipleOf": 3
}
```

```
<!LANGUAGE pt>
{
  DFJS_NOT_OBJECT: gen => { return gen.integer(0,33) * 3 }
}
```

```
18
```

Figure 20: Non-oject *JSON* Schema and respective *DSL* model and example instance.

Generic Keywords

The first step is to check if the intended value must belong to a fixed set of values, i.e. the usage of any of the keywords const or enum which, at this point, already reflect the absence of any values disallowed with not. If any of these are present, the remaining keywords are ignored and the *DSL* string is generated from these alone. In case of both keywords, const takes precedence over enum, as it defines that the value is constant and immutable. The output *DSL* string produced from these keywords is a random choice from all of their respective values (typically, const will map to a single value, but if the user composes multiple instances of this keyword in the same schema, it will be treated as an alternative between several).

If no fixed set of values is defined, the solution checks if the value was specified with an interpolation function of *DataGen*, through the custom keyword _datagen, which will be explained ahead in 4.3. In this case, the remaining keywords are also ignored, since this feature's function is to overwrite the schema's content with the result of *DataGen*'s function. Otherwise, it goes on to actually translate the type's keywords. Data types **null** and **boolean** are basic, since they possess no specific keywords. These and the previously addressed keywords are translated as follows:

```
{ "enum": [ "red", "green", "amber", null, 42 ] }
```

```
gen => { return gen.random(...["red", "green", "amber", null, 42]) }
```

```
{ "const": "United States of America" }
```

```
gen => { return gen.random(...["United States of America"]) }
```

```
{ "_datagen": "fullName" }
```

```
'{{fullName()}}'
```

```
{ "type": "boolean" }
```

```
'{{boolean()}}'
```

```
{ "type": null }
```

```
null
```

Figure 21: Translation of generic keywords, _datagen, and data types boolean and null to the *DSL*.

String Type

The string type only has four different keywords: pattern, format, and minLength/maxLength . Only in the case of this data type, it was decided that there would be an order of precedence to its keywords since, realistically speaking, they will very rarely be used together (except for both length keywords): for example, it does not make much sense to define a string value according to a regular expression or format and then further constrain its length, as the former keywords already establish a very rigid template.

As such, if the schema has the keyword pattern, that will be the one to be translated, followed by format and, finally, the length keywords. While pattern and length keywords translate directly to DataGen's interpolation functions, format is more contrived, since there is a need to program a different *DSL* string for each possible format, in order to generate an according value, some of which map directly to existing interpolation functions and others that do not. For most of the latter, the program uses *DataGen*'s pattern function to generate random, valid values according to their templates (e.g. formats email and ipv4). Some generated model examples for keywords of this data type are presented below:

```
{
    "type": "string",
    "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

```
'{{pattern("^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$")}}'
```

```
{ "type": "string", "format": "email" }
```

```
'{{pattern("[a-z]{5,20}@(gmail|yahoo|hotmail|outlook)\.com")}}'
```

```
{ "type": "string", "format": "time" }
```

```
'{{time("hh:mm:ss", 24, false, "00:00:00", "23:59:59")}}'
```

```
{
    "type": "string",
    "minLength": 5,
    "maxLength": 30
}
```

```
'{{stringOfSize(5, 30)}}'
```

Figure 22: Translation of string type schemas to *DataGen*'s *DSL*.

Numeric Types

As for numeric types, there are five keywords: multipleOf, minimum, exclusiveMinimum, maximum, and exclusiveMaximum. The constraints on numeric types can get a lot more complicated if the user specifies, via schema composition keywords, that the value must be a multiple of several numbers simultaneously and/or not a multiple of one or more values. Let's first consider simpler cases where the not keyword is not used.

If the instance must be multipliable by one or more values, the program starts by calculating the *Least Common Multiple* (*LCM*) of all these numbers. This way, it is possible to consider a single value for the multiplicity of the instance, since the *LCM* and its multiples are the easiest way to obtain any number simultaneously multiple of all the original values. If the type in question is an integer, this is also taken into account before determining the *LCM*, by considering that the instance must also be a multiple of 1.

Next, the range keywords are evaluated, if present. The program determines the biggest and smallest integers that it is possible to multiply by the *LCM* (or 1, if no multipleOf constraint exists), let's call them ***LCM* range delimiters**, in order to obtain values that belong to the intended range. This is doable by implementing the following algorithm:

$$\begin{aligned} lower\_delimiter &= Math.floor(\frac{maximum}{LCM}) \\ upper\_delimiter &= Math.ceil(\frac{minimum}{LCM}) \end{aligned} \tag{1}$$

where the maximum and minimum, if specified via the exclusive keywords, are determined by offsetting their values by a small margin, to ensure that the exclusive limits are not included in the generateable range. If the schema only has either an upper or lower range boundary, the other *LCM* range delimiter is calculated by offsetting the existing one by 100 units, to provide confortable margin for values to be generated, e.g.:

$$\begin{aligned} lower\_delimiter &= Math.floor(\frac{maximum}{LCM}) \\ upper\_delimiter &= lower\_delimiter + 100 \end{aligned} \tag{2}$$

At this point, there are three different possible outcomes:

- if only the type is specified and nothing else, the program simply generates a *DSL* string for a random integer or float, accordingly;
- if no range keys are used (the only type-specific is multipleOf), the *DSL*'s interpolation function with the same name is used to generate a multiple of the *LCM*;

- if both range and multiplicity constraints are present, the value is calculated by randomly choosing an integer between the predetermined *LCM* range delimiters and multiplying it by the *LCM*.

```
{ "type": "number" }

{
    "type": "number",
    "multipleOf": 7
}

{
    "type": "integer",
    "multipleOf": 2.5,
    "minimum": 0,
    "exclusiveMaximum": 30
}
```

```
'{{float(-1000, 1000)}}'

'{{multipleOf(7)}}'



gen => { return gen.integer(0,5) * 5 }
```

Figure 23: Translation of numeric type schemas to *DataGen*'s *DSL*.

However, the negation of the keyword multipleOf makes these use cases a lot more contrived, since there is a necessity to further restrict the set of produceable values. As such, the *DSL* string produced in this occasion is different for all the alternatives above. Only one variant will be explained, since the same logic applies across all others:

```
{
    "type": "number",
    "multipleOf": 1.5,
    "minimum": 0,
    "exclusiveMaximum": 50,
    "allOf": [
        { "not": { "multipleOf": 6 } },
        { "not": { "multipleOf": 9 } }
    ]
}
```

```
gen => {
    let multiples = gen.range(0,34).map(x => x * 1.5);
    let final_multiples = multiples.filter(m => [6,9].every(x => m%x != 0));
    return !final_multiples.length ? gen.random(...multiples) : gen.random(...final_multiples);
}
```

Figure 24: Translation of a complex numeric type schema to *DataGen*'s *DSL*.

As seen in the preceding image, the schema has impositions on the range of the value, as well as what it must and must not be a multiple of. This translates to the *DSL* via a *JavaScript*

anonym function, where *DataGen* first calculates all valid multiples in the designated range and stores them in an array, then removing all elements that are multiples of unallowed numbers. The final value is selected randomly from the alternatives with the interpolation function random, however the program firstly does a safety check to ensure that the final array is not empty: if that is not the case, then it is impossible to produce a value that obeys all restrictions specified in the schema, so it simply selects a regular multiple in the range from the first array.

Object Type

The set of type-specific keywords for objects contains properties and patternProperties, to specify property schemas according to their key; additionalProperties and unevaluatedProperties , for validating unspecified properties; required, to make properties mandatory, propertyNames , to validate the object's keys, and finally size keywords (minProperties and maxProperties).

DataGen From Schemas also treats dependentRequired (used to require properties based on the presence of certain keys) and dependentSchemas (to apply subschemas if certain properties exist) as object-specific keywords - for every new property selected for the final instance along the pipeline, required or not, the solution checks these keywords for dependencies. In case of the former, if any properties are dependent on the newest key, these are also translated and added to the object. As for the latter, if there is a subschema dependent on the latest key, it is parsed and used to extend the current object schema.

For this type, the instance's model is firstly prototyped in a **temporary object**, mapping each key to the *DSL* string of their respective value, in order to more easily manage all the different properties that may be produced. Only after determining all properties does the program produce a model for the whole object, from this *DSL* string map.

The first operation executed by the program is determining a random size for the final object, taking into account all relevant factors such as minProperties and maxProperties, required properties, and the permission or not of unspecified properties (additionalProperties , unevaluatedProperties). The calculated size will always necessarily allow room for at least the required properties, and if only the object type is specified and no other keywords are used, the program will generate an object with between 0 and 3 properties, where both the key and value are random.

Then, *DataGen From Schemas* iterates the required properties and generates an according *DSL* string for each of their values' schemas, storing the pair in the temporary map. Once the required properties have been produced, the solution executes the following pipeline sequentially, until it reaches the designated size for the final instance:

- iterates the non-required properties specified in properties sequentially, producing *DSL* strings for their respective values and storing the pairs in the map. In case the

object instance needs more properties than those that are required, it makes more sense to produce other properties specified by the user before random ones;

- iterates the value of the keyword patternProperties - for each of its properties, there is a probability (customizable by the user) to produce, at most, one according instance property (to avoid possibly creating repeated keys and messing up the property count), where the name of the property is obtained through a regular expression value generator;

- if additional properties are not allowed or not explicitly mentioned in the schema, the intermediate structure is considered finalized with the properties it currently has. Do note the mention of **explicitly** allowing additional properties - if neither of the keywords additionalProperties and unevaluatedProperties are specified, then the user most likely wants an instance with only the properties covered by the keywords properties or patternProperties - as such, it would not make much sense to generate other random properties, just because it is not explicitly disallowed;

- if the schema specifies additional properties, additionalProperties has precedence over unevaluatedProperties, so if both keywords are present, additionalProperties's schema prevails, else it is the only used keyword's. The program translates this schema into a *DSL* string that it uses to generate values for additional properties with random names, which are obtained either according to the propertyNames schema, if present, or by generating small chunks of *lorem ipsum*. It does this until it reaches the intended key set size for the instance.

With all the selected properties' *DSL* strings, the application joins them together in a string encased by curly braces, in the syntax of a regular *JavaScript* object that *DataGen* is able to parse and then use to generate an according dataset.

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" },
    "type": { "enum": [ "residential", "business" ] },
    "residents": {
      "type": "object",
      "propertyNames": { "pattern": "inhabitant[1-3]" },
      "additionalProperties": { "enum": [ "John", "Ian", "Mary" ] },
      "minProperties": 2,
      "maxProperties": 3
    }
  },
  "required": [ "street_address", "city", "state", "type", "residents" ]
}
```

```
{
  street_address: '{{stringOfSize(0, 100)}}',
  city: '{{stringOfSize(0, 100)}}',
  state: '{{stringOfSize(0, 100)}}',
  type: gen => { return gen.random(...["residential","business"]) },
  residents: {
    inhabitant2: gen => { return gen.random(...["John","Ian","Mary"]) },
    inhabitant3: gen => { return gen.random(...["John","Ian","Mary"]) },
    inhabitant1: gen => { return gen.random(...["John","Ian","Mary"]) }
  }
}
```

Figure 25: Translation of an object type schema to *DataGen*'s *DSL*.

Array Type

There are several keywords for this data type: items and prefixItems, the main way to evaluate items; additionalItems and unevaluatedItems, for those that do not validate against the former keywords; contains, minContains, and maxContains, for inclusion of items with certain schemas; finally, length keywords (minItems, maxItems), to restrict the array size, and uniqueItems, to determine the uniqueness of items in the instance.

For this data type, the program creates a **temporary array structure** firstly, in order to malleably plot the intended items for the instance, and only at the end converts it to a *DSL* string, similar to how it operates with the object type. The solution keeps track, at all times, of the allowed number of items, which can be established with minItems and maxItems. If the maximum amount of items is ever reached (and also never before the minimum is attained), the program stops the execution of the pipeline that will be described ahead and produces the *DSL* string from the items it has at that point.

The first step of the algorithm is to check for prefixed items in the schema: if any were specified, the program sequentially generates their respective values' *DSL* strings and pushes these to the temporary array, in order.

Then, *DataGen From Schemas* parses the inclusion keywords. In the workflow of this solution, these keywords effectively have a **dynamic semantic**, since it is impossible to validate the array for the inclusion of elements with the specified schema before generating it, even in the *DSL* model. As such, a compromise was needed to adapt these keywords: after parsing the prefixed items, if more items are allowed in the array, then the solution will push the necessary amount of new elements with the structure of the contains schema (one if minContains and maxContains are not used, otherwise a random number in the indicated range). Note that if the intent of the user is for the inclusion keywords to validate some of the prefixed items, then the program cannot guarantee this.

Once all the "hard-coded" elements have been parsed, the program then checks for other items. The keyword additionalItems has precedence over unevaluatedItems, so if additional items are allowed and both keywords are specified, additionalItems's schema is considered, otherwise it is the only specified keyword's. If the user also disallowed any instance of the keyword contains, its schema is inverted and used to extend the schema for additional items, in order to guarantee that all additional items wholly conform to the user's configuration, both what they should and should not validate against. This final schema is used to sequentially generate a random amount of *DSL* strings for additional items, within the intended array size, which are appended to the temporary structure.

Finally, *DataGen From Schemas* checks if all these items must be unique (uniqueItems). This keyword, as is the case with the inclusion keys, demands a compromise and an intelligent workaround, since it is impossible to check for the items' uniqueness before generating them, and is the reason that warrants the usage of a temporary array to store the *DSL* string of

each element, instead of simply building a single string from the start and appending each item to it. As such, depending on the value of this keyword, two different types of *DSL* strings may be created for the final array:

- **non-unique items** - if either the keyword is not specified or its value is false, then the procedure is to simply create an array string, where its elements are the *DSL* strings stored in the temporary array structure maintained along this pipeline. *DataGen* can interpret this syntax and generate all elements according to their respective model;

```
{
  "type": "array",
  "prefixItems": [
    { "type": "number", "minimum": 0, "maximum": 100 },
    { "type": "string" },
    { "enum": ["Street", "Avenue", "Boulevard" ] },
    { "enum": ["NW", "NE", "SW", "SE" ] }
  ],
  "items": { "type": ["string", "number"] },
  "contains": { "type": "integer" },
  "minItems": 6,
  "maxItems": 8,
  "uniqueItems": false
}
```

```
[
  gen => { return gen.integer(0, 100) * 1 }
  '{{stringOfSize(0, 100)}}',
  gen => { return gen.random(...["Street", "Avenue", "Boulevard"]) },
  gen => { return gen.random(...["NW", "NE", "SW", "SE"]) },
  '{{integer(-1000, 1000)}}',
  '{{stringOfSize(0, 100)}}',
  '{{float(-1000, 1000)}}',
  '{{float(-1000, 1000)}}'
]
```

Figure 26: Translation of an array type schema with non-unique items to *DataGen*'s *DSL*.

- **unique items** - in this case, a *DataGen* anonym function is used to implement an algorithm that attempts to generate an array where all elements are distinct. As illustrated below, the procedure is the following: whenever *DataGen* creates the next item, it checks if the array already contains the new value. If not, it pushes the element to the array and moves on to the next one. Else, it generates the same item again (which can produce a different result, since the *DSL* strings accomodate margin for randomness), for a maximum of 10 tries per item. In case none of them produces a new value, the result of the latest try is pushed to the array anyway, to prevent crashing the program, and the resulting array ends up not obeying the uniqueItems restriction.

```
{
    "type": "array",
    "prefixItems": [
      { "type": "number", "minimum": 0, "maximum": 100 },
      { "type": "string" },
      { "enum": ["Street", "Avenue", "Boulevard" ] },
      { "enum": ["NW", "NE", "SW", "SE" ] }
    ],
    "items": { "type": ["string", "number"] },
    "contains": { "type": "integer" },
    "minItems": 6,
    "maxItems": 8,
    "uniqueItems": true
}
```

```
gen => {
    let arr = [];
    for (let i = 0; i < 7; i++) {
        for (let j = 0; j < 10; j++) {
            let newItem
            if (i==0) newItem = gen.integer(0, 100) * 1;
            if (i==1) newItem = gen.stringOfSize(0, 100);
            if (i==2) newItem = gen.random(...["Street", "Avenue", "Boulevard"]);
            if (i==3) newItem = gen.random(...["NW", "NE", "SW", "SE"]);
            if (i==4) newItem = gen.integer(-1000, 1000);
            if (i==5) newItem = gen.float(-1000, 1000);
            if (i==6) newItem = gen.stringOfSize(0, 100);
            if (!arr.includes(newItem) || j==9) {arr.push(newItem); break;}
        }
    }
}
```

Figure 27: Translation of an array type schema with unique items to *DataGen*'s *DSL*.

As a by-product of this algorithm, another compromise arises: *DataGen From Schemas* can only attempt to generate arrays with unique items **if all items are elementary**, i.e. neither objects nor arrays. This is the case because *DSL* strings for complex values are bigger, more convoluted and impossible to incorporate in a *DataGen* function syntax, in the same way a basic interpolation or anonym function can be. To summarize, *DataGen From Schemas* is not the best tool to adapt the keyword uniqueItems in specific, due to its ill-suitability to the workflow and *DataGen*'s *DSL*, however it still manages to satisfy some use cases.

4.1.5 *Schema Inverter*

In order to be able to integrate the restrictions of the keyword not into the logic of the resulting instance, *DataGen From Schemas* employs a schema inverter capable of manually fabricating a schema complementary to not's, at least partially, so that no value ever validates against both simultaneously. Then, the application can extend the instance's remaining schema with this new inverted one and delete the property not, producing the same effect.

To start off, not has some interesting use cases related to the disallowance of entire data types, which must be taken into account:

- **universal schema** - a true schema (true or {}) validates any kind of value, of any type. Thus, the prohibition of such a schema with not does not make sense, since it means that it is not possible to generate any value from it, so the application throws an error in this case, warning the user;
- **schema with only type** - disallowing a schema with only this keyword is equivalent to prohibiting the specified data type(s). As such, the program maintains an array with the forbidden types along the pipeline, to ensure that it does not produce any such value. Disallowing all types of values is exactly the same as using not with an universal schema, so the application also throws an error in that case.

Generic Keywords

Keywords const and enum indicate that the instance is either a certain imutable value or one of several hard-coded alternatives. Their negation implies that the instance can never be any of the listed values, so the program maintains an array with every disallowed value and ultimately checks that these are not options when generating the *DSL* model from the final, simplified schema. If the instance's remaining schema possesses any of these two keys without negation, the application checks their values and removes any prohibited alternatives that it finds, also removing the keyword itself from the schema if it is left with no valid values.

String Type

In order to invert a string pattern, the program inverts the regular expression in the following manner:

$$\begin{aligned} |pattern| &=> \hat{}(?!.^*|pattern|) \\ E.g. : [a\text{-}zA\text{-}Z]+ &=> \hat{}(?!.^*[a\text{-}zA\text{-}Z]+) \end{aligned} \tag{3}$$

where ^ makes the regular expression start matching at the start of the string, (?! starts a negative look-ahead, which looks ahead in the string and does not permit a match against the pattern in question, $.^*$ translates to zero or more characters of any kind (except line terminators), i.e. the match can be anything except a string that matches the disallowed pattern, and) ends the negative look-ahead. The resulting regular expression is incapable of producing any value that matches with the original pattern.

As for the keyword format, the application also maintains a separate array for disallowed string formats and checks the instance's remaining schema for this keyword - if it is present and its value has been disallowed, it is removed from the schema. As such, if a string schema specifies a given format which is later disallowed, the program produces a string according to its remaining schema restrictions or randomly, if there are none.

Finally, the size keywords: if maxLength is present in the not schema, the program sets minLength as its incremented value in the instance's schema (the opposite of a string with **n** characters is a string with at least **n+1** characters). This solution is valid whether or not the not schema also specifies a minimum length (**n+1** characters is also outside the range between **n-m** and **n** characters); if not **only** specifies minLength, either the program sets the maximum length on the instance's schema as the decremented value of the prohibited keyword, if the minimum length is positive, or throws an error if it is zero, warning the user that it is impossible to generate a string with negative length.

Numeric Types

If the schema is of integer type, then *DataGen From Schemas* changes its type to number and explicitly prohibits the generation of integers.

As for the range keywords, this data type does not have the non-negative restriction of the string type's size keywords value range. Thus, it is possible to invert only one of the range keywords and ignore all others to create mutually exclusive schemas. Additionally, if a schema has both maximum and exclusiveMaximum, or both of their counterparts, the program deletes the redundant one (e.g. having a maximum value higher than an exclusive maximum is redundant, since no value higher than the latter will ever be produced).

If the not schema specifies the keyword minimum, the program deletes all range keywords (including this one) and sets exclusiveMaximum with the same value. This is complementary for both integer and fractionary values. If not, it checks for exclusiveMinimum next, deleting the range keywords and setting a maximum with the same value. It repeats this process for maximum (replaced with exclusiveMinimum) and then exclusiveMaximum (swapped with minimum).

If the schema does not possess range keywords, the application looks for multiplicity constraints, maintaining an array with all values that the instance must not be a multiple of. It then checks the instance's remaining schema for the keyword multipleOf and, in case it specifies a disallowed value, removes it from the schema.

Object Type

In this data type, the inversion of the schema becomes recursive, given that a lot of the object-specific keywords' values are subschemas. As such, for both keywords properties and patternProperties, the application iterates their key set and recursively calls the schema inverter for each of their values. This way, each specified property will have a new, complementary schema, e.g. if a property was meant to be a string with maximum length **n**, it will instead become a string with minimum length **n+1**.

This approach was chosen in favour of disallowing properties with the specified keys, as this would require the maintenance of yet another external array with the prohibited keys and results would become a lot more random, generating properties with new, arbitrary keys. The schema inverter is also called recursively for the keywords additionalProperties, unevaluatedProperties and propertyNames, whose value is a single subschema.

As for the keyword required, the program maintains an array with all the unrequired properties and later checks the instance's remaining schema for this keyword: if specified, it checks its values and removes any unrequired keys present. In the end, if the keyword's value becomes an empty array, the keyword itself is removed from the schema.

Finally, the inversion of the object size keywords follows exactly the same algorithm that is used with their string type equivalents (4.1.5).

Array Type

The application calls the schema inverter recursively on several keywords of this data type: items and unevaluatedItems, whose value is a single subschema that must be inverted; prefixItems, which defines an array of subschemas, so the application iterates this array, obtaining complementary schemas for each of the instance's prefixed items.

The value of the keyword uniqueItems is just a boolean, so the program simply changes it to its opposite, and the string type size keywords' algorithm is yet again implemented with this type's minItems and maxItems.

Finally, for the containment keywords:

- if the not schema specifies an upper limit, the program sets minContains in the instance's remaining schema as its incremented value and removes maxContains, if present. If it specifies a lower limit, the procedure is the opposite;
- if it only specifies the keyword contains, the application first removes any types present in its schema without further type-oriented keywords, thus prohibiting those data types; next, it calls the schema inverter recursively on the keyword's schema, obtaining its complementary schema for different items that must now be included in the final instance.

The other alternative for the inversion of this keyword was to generate a complementary schema and prohibit any value compliant with it from occuring in the instance. However this option was a lot more complicated and harder to achieve, potentially interfering with other keywords of the array schema in unexpected ways. Since the concept of inverting a schema is open to interpretation and its final objective was established, in this project, as simply creating a schema with no validating instances in common, it was decided to instead force the array to contain elements compliant with the inverted contains schema.

4.1.6 *Schema Extender*

A schema extender was developed to manually extend a base schema with each new type-specific keyword. For each data type, the new keyword is compared to the already existing ones and incorporated in a reasonable way. This solution handles each individual *JSON* Schema keyword differently, as they all have different meanings. If the base schema does not have the same keyword that is being used to extend it, it is simply assigned to the structure. Otherwise, the logic applied with each keyword is as follows:

Generic Keywords

For the generic keywords const and enum, the old and new values are concatenated in an array. This is logical for enum, since it indicates all possible alternatives for the instance. As for const, it does not really make sense to declare more than one constant value in a schema, so it was decided to make them alternatives in the event that the user did so, of which ultimately only one will be selected, to make the program more malleable.

String Type

Regarding string-specific keywords, regular expressions of the keyword pattern are AND'ed together, using non-consuming expressions, (?=*expr*). These expressions continue matching

from the original match-point after matching each pattern, ensuring that the string is compliant with all the patterns, i.e.:

$$\begin{aligned} |pattern1| \;\&\&\; |pattern2| &=> (?=|pattern1|)(?=|pattern2|) \\ E.g. : [a\text{-}zA\text{-}Z]+ \;\&\&\; [0\text{-}9]+ &=> (?=[a\text{-}zA\text{-}Z]+)(?=[0\text{-}9]+) \end{aligned} \tag{4}$$

As for the keyword format, the base format is overwritten by the new one, since their templates are mutually exclusive.

For the string size keywords, the following algorithm is applied: when extending minLength, if the base schema does not have the same keyword, it is simply assigned to it, otherwise the base value is overwritten only if the new value is higher than the former (having a minimum length of **n** is less restrictive than having **n+m**). After this, the program checks for the keyword maxLength in the schema: if it exists and its value is lower than minLength's, maxLength is erased from the structure, for the sake of coherency. The extension of maxLength is analogous, but with the opposite logic (only overwrites if the new value is lower, then compares with minLength).

Numeric Types

The extension of multipleOf follows the same logic applied to generic keywords - it makes sense to concatenate the values of all instances of this keyword, as what the schema is indicating is that the value must be a multiple of several numbers simultaneously.

As for the range keywords, these are extended similarly to the string size keywords, with the extra exclusive use cases: e.g. when extending the keyword minimum, if the base schema has a lower exclusiveMinimum, this keyword is removed from the schema and minimum is assigned, otherwise exclusiveMinimum prevails and minimum is discarded (again, the most restrictive keyword prevails). The application then checks for coherency not only with the keyword maximum, if present, but also exclusiveMaximum, also deleting it if it has a value lower or equal to minimum's. The structure only ever stores one keyword for the same boundary, since having both repeats information unnecessarily. The three remaining range keywords are extended analogously.

Object Type

This data type possesses some keywords whose method of extension is open to interpretation, since there is more than one sensible way in which one could go about this procedure. For keywords with more than one alternative, users can specify their preference in the application's settings, which provide a set of customizable options for cases such as these.

Keywords properties and patternProperties define objects where each key is the name of an instance property and each value is the schema used to validate that property. To start off, when extending a base schema that possesses one of these with another instance of the same keyword, each of the new properties with a key not found in the base keyword's object is assigned to it. If a key is repeated, however, there are two possible approaches: the default behaviour is to extend the base key's schema with the new one's, merging them together to create a complementary schema with the restrictions of both, but it is also possible to simply overwrite it, in which case the validating schema of the property in question becomes the new one.

As for keywords additionalProperties, unevaluatedProperties, and propertyNames, there are also both of these options, extending or overwriting, since each of them has a value of a single subschema. Thus, with these keywords, as well as the previous ones, the default procedure is to recursively call the schema extender within its own routine, for their values, in order to generate more comprehensive schemas for their respective instances.

Finally, when extending required, the program simply concatenates the values of the base and new instances of the keyword, bundling all required properties of the object instance together without any duplicates. As for the size keywords, minProperties and maxProperties, the schema extender implements exactly the same algorithm described for their string type equivalents (4.1.6).

Array Type

These type's items and unevaluatedItems keywords behave exactly as the object type's keywords for additional properties do, which makes sense given that they are each other's counterparts, for different data types, and their value is a single subschema. As such, the value of the base instance of each of these keywords can either be extended with its new one, by calling the schema extender recursively, or overwritten by it, according to the user's preference.

The keyword prefixItems specifies an array of schemas, depicting the first elements of the array instance. It does not make much sense to extend an existing prefixItems with another, but the user is given several choices in case they do so, in order to better suit their intentions: the default behavior is to, **for each common index, extend the base schema with the new one**, invoking the program recursively for each index's values. However, it is also possible to execute a **partial overwrite**, overwriting the base keyword's schemas with their new correspondents, only for those at shared indexes, a **total overwrite**, completely replacing the base keyword's value with the new one (even if their lengths differ), or to simply **append the new array of prefixed items' schemas to the base one**. Also, in the two first options, the program either leaves the remaining base items' schemas that do not have corresponding

new ones at the same index as are, if the base keyword's array is bigger in size, or appends the remaining new schemas, if the new keyword's array is bigger.

As for contains, the application merges both instances of the keyword together, extending one with another, since the extension of this keyword intuitively means that the array must obey all schemas specified in instances of the keyword simultaneously.

For both pairs of size keywords - minItems/maxItems and minContains/maxContains -, the schema extender follows the same logic applied to their string type's counterparts (4.1.6) and, finally, when extending the keyword uniqueItems, the base value is overwritten by the new one, since these are booleans and, therefore, their possible values are exclusive alternatives.

4.1.7 *Settings*

It was concluded from the analysis of related work that it would be valuable for a program such as *DataGen From Schemas* to have a settings menu with customizable options, allowing the user to have further control over certain specifics of the schema dialect that can become tricky when reversing its normal workflow and generating data from the formal specifications instead.

This idea was further explored during the implementation of the solution, evaluating which particularities it would be useful to have the ability to influence directly from outside the schema and, as such, the following options were centralized in a settings menu that is made available to the user for *JSON* Schema:

- **recursion** - the user can define upper and lower limits for the recursion of structures in a given schema, ensuring that these structures present a minimum established level of nesting and do not exceed a certain maximum;
- **if keyword validation probability** - as mentioned in 4.1.3, unless the schema value of this keyword is explicitly true or false, the program will decide if it validates the schema or not based on a probability (50/50% by default). This probability can be altered by the user, in order to influence the odds of this decision;
- **patternProperty keyword generation probability** - *DataGen From Schemas* generates, at most, a single property from each pattern property, in order to avoid creating properties with the same name that overwrite each other. The probability of generating such a property according to its key's regular expression is usually 80%, but users can also modify this to their liking;
- **ability to generate unspecified additional properties** - typically, if the user creates an object schema without using the keywords additionalProperties or unevaluatedProperties , the intent is to produce only the properties explicitly declared in the schema. However,

DataGen From Schemas could theorically generate others randomly, since additional properties are not disallowed, as long as the established object size is respected. Thus, the user is given to possibility to enable or disable (default) this option;

- **extension of schemas** - as was described in 4.1.6, the application defines a certain method for extending each different type-specific keyword of the syntax. Some of these are more convoluted and leave room for interpretation, as there are several possible approaches, for example how to proceed when extending instances of the keyword prefixItems. As such, the user is given the ability to choose the method that best suits their needs in these cases, specifically when extending properties and patternProperties , keywords with a schema value (propertyNames, additionalProperties, unevaluatedProperties , items and unevaluatedItems), and the aforementioned keyword prefixItems.

4.2 XML SCHEMA COMPONENT

The infrastructure required for this component of *DataGen From Schemas* is very similar to the *JSON* Schema component's, with a parser to analyze the schema and extract information, as well as a translation program to produce a *DSL* model from said information.

The *XML Schema Definition* (*XSD*) language has a very different way of moduling data instances from *JSON* Schema, both in syntax and content. For starters, it has a lot of redundancy that complicates its parsing: e.g. while in *JSON* Schema, each value's type is clearly declared or implied in its schema, in *XSD* the type can either be defined as an attribute of the element or specified via a simpleType/complexType element nested within it. Another example is how to specify the data types that a union can select from: these types can all either be stated in the element's memberTypes attribute, nested inside the element with type elements or both simultaneously. This redundancy introduces an extra layer of complexity in the grammar, as it will need to be able to filter all different ways of approaching certain cases like this.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="internationalShoeSize">
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="sizing" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:element name="myShoeSize" type="internationalShoeSize"/>
</xs:schema>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="myShoeSize">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:decimal">
          <xs:attribute name="sizing" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 28: Redundant *XML* schemas.

Furthermore, types are not as clear-cut as in *JSON*, where there are six different possibilities and schemas can only ever be restricted with type-specific keywords or tools of schema composition, so the set of operations that each type accomodates is relatively limited. In *XML* Schema, data typing is far more contrived: first of all, the syntax distinguishes between **simple types** and **complex types**, which are not interchangeable, i.e. certain elements can only have a simple type and others a complex type, and some both. Additionally, complex types can either have **simple content** or **complex content**, and the set of operations performable on each is radically different. Next, the alteration of a type implies the usage of a lot of different nested *XML* elements and there are two kinds of operations: extension and restriction . These have different purposes and syntaxes, which also differ according to the type's simple or complex nature and, in the case of a complex type, the nature of its content. For reference, there are three different restriction elements, appliable to elements simpleType, simpleContent, and complexContent, and all of them have a distinct set of nestable elements. As such, types in *XSD* are a lot more detailed and structured, which implies a much more complex and sturdy intermediate structure and semantic grammar validation, for tracking of (legal) transformation of different types.

Once it became clear that parsing *XML* schemas would be a lot more difficult than their *JSON* counterpart, the first step was to conduct a thorough analysis of the syntax and determine which elements and attributes were worth parsing and which were not, taking into account that the *XSD* language possesses an overall much wider and varied syntax than *JSON* Schema.

The main references used to study the syntax were *Microsoft*'s XML Schemas (XSD) Reference and the *W3C* Recommendation, which represents the official source and publication of the *XSD* specification, since this language was developed by the *W3C* team. *Microsoft*'s reference is much cleaner, carefully detailing *XML* Schema elements, data types, etc. in a user-friendly and readable interface, but lacks in some areas, e.g. the data type facets which are not explained and simply link to the *W3C* manuals. Meanwhile, *W3C*'s reference is cluttered and harder to read, but minutely details every aspect of the language, namely lexical spaces of elements' instances and such.

4.2.1 *Grammar*

In order to prototype an efficient and feasible data generator from *XML* Schema, the first step was to establish a relevant subset of the syntax's elements, selecting those which are more common in schemas and useful for dataset specification. The grammar is capable of recognizing every *XSD* element, which was implemented to prevent it from crashing or halting the execution of the pipeline because of the presence of elements that are not of

interest, such as annotations (used to document schemas). Such elements are filtered but ignored, not contributing in any way to the final instance.

Figure 29: *XML* Schema elements.

The *XML* Schema component was not projected to support cross-schema referencing, unlike its *JSON* counterpart, as this entails complicated tasks like the parsing of different namespaces and prefixes. Cross-schema referencing would only be pondered at the end, depending on the remaining time available for the project, which ended up not being enough to implement this feature.

This grammar was developed with the same philosophy as the *JSON* Schema grammar and, as such, it possesses the same key features of the latter, described in 4.1.1: besides **validating the schema semantically** and **producing detailed error reports** for the user, it also **restricts each element's semantics and lexical space**, by configuring its respective attributes and acceptable nested elements (content), which differ from element to element, and constraining their lexical space minutely.

Let's look at an example that justifies this necessity: the native data types (figure 36) all have very specific lexical spaces, sometimes with only minor differences between each other (especially the derived string types). When restricting a type with the enumeration facet, it is very important to ensure that the enumerated value conforms to the norms of the type in question: this facet's value does not have a predefined lexical space, but instead inherits the one from the type in question. The grammar is able to keep track of this and enforce that space on the value introduced by the user, thus ensuring that the produced instance is coherent.

Another example is that, as mentioned before, there are three different restriction elements, so it is crucial to differentiate them and restrict their lexical spaces accordingly since, for example, the variants respective of simple types and complex types with complex content have the same set of attributes, but radically different content:

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema Namespace}...>
Content: (annotation?, (simpleType?, (minExclusive | minInclusive |
maxExclusive | maxInclusive | totalDigits |fractionDigits | length |
minLength | maxLength | enumeration | whiteSpace | pattern)*))
</restriction>
```

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema Namespace}...>
Content: (annotation?, (group | all | choice | sequence)?, ((attribute |
attributeGroup)*, anyAttribute?))
</restriction>
```

Figure 30: Definitions of the aforementioned restriction elements, respectively.

Implemented XSD elements

- schema is mandatory, to declare the namespace of the schema and respective prefix. Every schema must have this element in order to be valid;
- element is also mandatory for the program to be able to generate data from the schema: it must have at least one root element from which to generate the instance. Furthermore, it's the element used to define data units and essential to structure complex structures;
- following the previous point, **particles**, which are elements that can have occurrence attributes and always appear as part of a complex type definition or as part of a named model group. The particles implemented in the grammar (including element, which was already mentioned) were:
  - all - declares a group of elements that can appear in any order in the containing element;
  - choice - enables the presence of one and only one of the elements from the selected group in the containing element;
  - group - combines a set of element declarations into a reciclable group, to be incorporated into complex type definitions;
  - sequence - defines a set of elements that must appear in the specified order in the containing element.
- **attributes** - attribute and attributeGroup, to declare (sets of) attributes that can be incorporated in complex type definitions;
- **type elements** - simpleType and complexType, in order to specify an element's data typing/structure. Both of these elements require the usage of other associated elements to create their definitions:
  - **simple type definitions** - list (for specifying lists with whitespace-separated elements), restriction (to further constrain another simple type), and union (for elements that can be of multiple types);
  - **complex type definitions** - **particles** and **attributes** (which were already mentioned above), simpleContent and complexContent (to specify a complex type's

kind of content), extension (to extend complex type definitions), and restriction (to restrict complex type definitions).

Ignored XSD Elements

All of the remaining *XSD* elements that were not mentioned above are not acounted for by the grammar, for relating to one of three distinct concepts:

- **multiple *XSD* documents and namespaces** - any (particle) and anyAttribute, which serve to enable elements/attributes from other namespaces to appear in the instance, import, include, and redefine. This concept was not relevant, since it was already determined that *DataGen From Schemas* would only operate with the local namespace of the schema;
- **identity constraints** - field, key, keyref, selector, and unique, whose purpose is to enforce certain properties on elements and are very similar to database concepts such as **uniqueness**, **primary key**, and **foreign key**, specified through *XML Path Language* (*XPATH*) expressions. These elements are rarely used in *XML* schemas and require knowledge on *XPATH*, which the majority of casual users does not possess. As such, it did not seem immediately necessary to implement these elements in the schema, in view of the value they had to offer;
- **schema annotations** - annotation, which requires appinfo or documentation. These elements are used to annotate schema documents and leave important information accessible to future users of said schemas. Such as comments, these do not have any impact in the final instance, so the program simply ignores them.

By modifying diagram 29, it is possible to get a clearer look at the the final subset of *XSD* elements selected for implementation in the program, as well as those that were not of chosen:

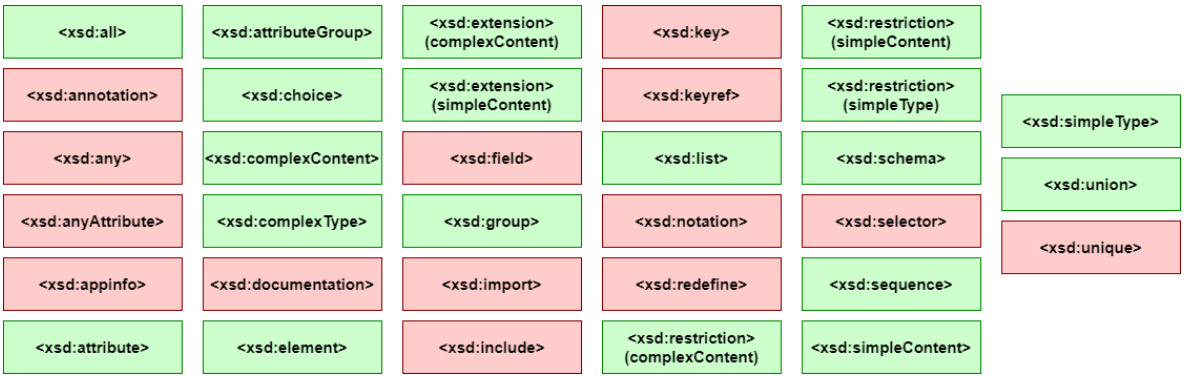


Figure 31: Selection of *XML* Schema elements for *DataGen From Schemas*.

4.2.2 *Intermediate Structure*

XML Schema's syntax and structure are fundamentally different from *JSON* Schema's, so it's natural that the type-oriented structure used in the other component is not suitable for this language. *JSON* Schema's most basic data unit is a *JSON* property, a simple key/value pair, which is easy to store as is since this format is universal throughout the schema, i.e. no keyword made available by the syntax deviates from this key/value standard. For this reason, the information extracted from a *JSON* schema requires minimal adjustments even after being reorganized in function of its generateable data type.

This is not the case of *XSD*'s syntax at all, given that its basic data unit, an element, has a lot more information that needs to be filtered and extracted, and such information is spread out over the element's structure, not centralized in a single value. Every element has an **asset type** (attribute, choice, element, etc - green), **attributes** (specific to each asset type - yellow) and **content** (nested elements, each asset type also has a specific set of nested asset types - orange):

```
<choice
  id = ID
  maxOccurs= (nonNegativeInteger | unbounded) : 1
  minOccurs= nonNegativeInteger : 1
  {any attributes with non-schema Namespace}...>
Content: (annotation?, (element | group | choice | sequence | any)*)
</choice>
```

Figure 32: Structure of an *XSD* element.

The intermediate structure projected to store information from *XML* schemas revolves around these three components - the application stores each *XSD* element's information in a *JavaScript* object with three properties, where **element** is its asset type, **attrs** holds its attributes as simple key/value pairs, since all attributes have simple content, and **content** is an array of objects that correspond to its nested elements, in the format described in this paragraph, e.g.:

```xml
<?xml version="1.0"?>
<xs:choice minOccurs="0" maxOccurs="unbounded">
  <xs:element name="elephant" type="xs:string"/>
  <xs:element name="bear" type="xs:string"/>
</xs:choice>
```

```json
{
  "element": "choice",
  "attrs": { "minOccurs": 0, "maxOccurs": "unbounded" },
  "content": [
    {
      "element": "element",
      "attrs": { "name": "elephant", "type": "xs:string", "maxOccurs": 1,
                 "minOccurs": 1, "abstract": false, "nillable": false },
      "content": []
    },
    {
      "element": "element",
      "attrs": { "name": "bear", "type": "xs:string", "maxOccurs": 1,
                 "minOccurs": 1, "abstract": false, "nillable": false },
      "content": []
    }
  ]
}
```

Figure 33: Example of an *XSD* element's intermediate structure.

This structuring makes it possible to store all information of an element in an organized and formulaic manner, which facilitates the structure's traversal and querying.

Attribute Referencing within the Schema

After storing all of the schema's information in the intermediate structure, the parser then resolves all existing references within the schema, which are specified through certain attributes. Two different attributes classify as references in this context: ref, which can be observed in the elements attribute, attributeGroup, element, and group, and points to entire elements and their structures, avoiding duplication of code; type, which can be observed solely in the elements attribute and element, and is used to specify their data typing by referencing simple or complex types, either built-in (in this case, only simple types) or declared elsewhere in the schema by the user.

XML Schema has a key characteristic that assists the resolution of these references: it is only possible to reference top level elements, i.e. elements declared at the root of the schema. The only exception is that of built-in simple types, referenced with the attribute type, which are native to the language and not explicitly declared in the schema. Thus, the program only needs to iterate the structure recursively, checking each element's attributes for ref or type. If it does not have any of these attributes, the solution proceeds to loop through its content, checking each of its nested elements and repeating this procedure. Otherwise, it identifies the referenced element at the root of the intermediate structure via its attribute name (which is required in top level elements) and assigns its attributes and content to the referencing element (the new values of repeated attributes will prevail over the original element's), thus centralizing all information in the structures of the schema's root elements of asset type element, from which the dataset can be generated. This way, the *DSL* translator will have all the necessary data in the structure of the element selected by the user for the dataset and

will be able to discard all other information filtered from the schema, allowing for a simpler and cleaner translation procedure.

However, the parser needs to perform another step before this, in order to be able to resolve all references, and that is to prepare all custom types declared in the schema, so that each of their structures possesses all of their respective data.

Parsing Custom Types

Type derivation is a very complex tool in *XML* Schema. Let's start with simple types: any custom simple type needs to reference a base type, which it either extends or restricts, and that base type must be compatible with the new type's content: the base type cannot be a complex type and each of the syntax's built-in types has a specific set of applicable constraining elements, i.e. data type facets, which are inherited by the derived type. For this reason, a simple type derived from a numeric built-in type cannot be restricted with the constraining facet maxLength, for example.

As for complex types, it is important to distinguish the two different kinds that exist, depending on the nature of their content:

- **simple content** - their content is based on a simple type (either built-in or custom) and the element can possess attributes, which normal simple types cannot;
- **complex content** - their content is a set of nested elements and the containing element of this type can also possess attributes.

As such, definition of complex types is even more complicated: simple content types must always be based on another type (using the element simpleContent), which can be either simple or complex with simple content, inheriting its content and also its attributes, in case of the latter; however, it is possible to define complex types with complex content from scratch, structuring them with particles and attributes, thus creating new complex structures that all elements of such types must abide by. Still, the user can also create types with complex content by deriving other such complex types (with the element complexContent), creating new complex types that further restrict or extend their bases, where each of their nested elements can be of any of the types mentioned. All of this is illustrated in the following image:

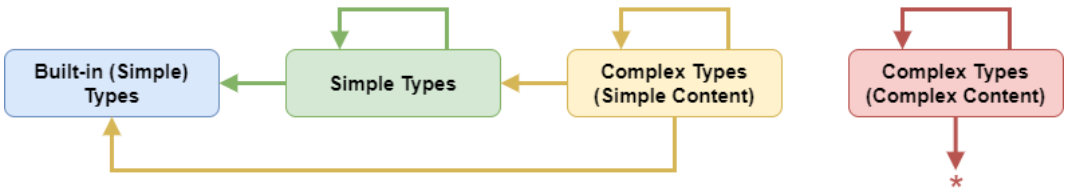


Figure 34: Flowchart of type derivation in *XML* Schema.

This diagram has two great implications:

1. the parser can only derive and finalize custom data types after analyzing the entire schema, because of their references to base types, since *XSD* documents do not establish any kind of mandatory order for the elements at the top level of the schema. As such, a custom type may reference a base type that is defined anywhere in the schema, so it is not viable to parse custom types as the parser finds them, since their base may not yet have been reached;

2. a schema may possess a vast web of interconnected type dependencies, so the order in which custom types are parsed is incredibly important, since it is necessary to ensure that a custom type's dependencies have all already been finalized by the parser when trying to resolve it, otherwise it will not be able to do so.

Concluding, it is crucial to determine the right order in which to parse the schema's custom types, but the schema itself does not help to determine this order in any way, due to its free placement of top level elements, resulting in a group of unresolved types after analyzing the entire schema, possibly dependent on each other.

TYPE SORTING ALGORITHM

The answer found to this dillema was to implement an algorithm with queues that loops through all of the remaining custom types and iteratively resolves them, as soon as their dependencies are complete.

There are two queues: one for types with simple content (simple and complex types) and another for complex types with complex content. While analyzing the schema, the parser immediately caches any custom type in the respective queue, whenever it finds one. At the end, each queue is complete with all the custom types of the schema that have yet to be resolved.

To determine the order in which these types must be resolved, it is crucial to take into account the subliminal hierarchy of type dependencies that exists in *XML* Schema, which figure 34 suggests:

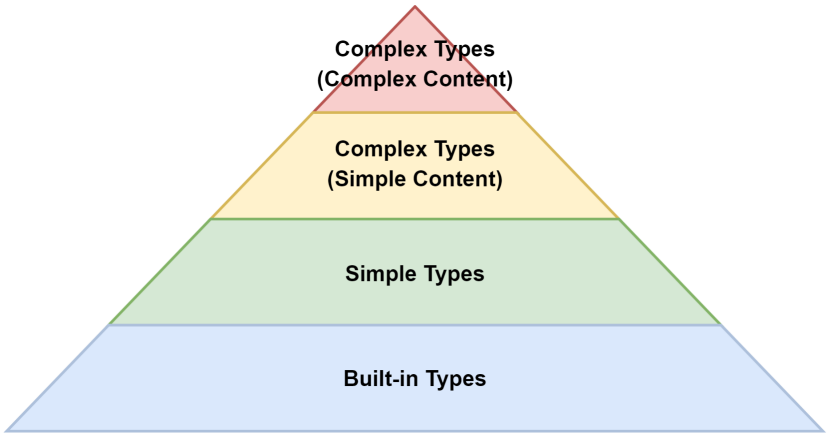


Figure 35: Hierarchy of type dependencies in *XML* Schema.

With this, it becomes clear in which order the custom types of the schema must be resolved - the language's built-in simple types are the only data types there are inherently finalized at any moment and do not require any additional parsing. Since any custom type with simple content must be based on another simple type, the lowest level derived types of a schema will necessarily be derived from the built-in types - these are the custom types that need to be parsed first, since their base type is already complete. Then, the derivation flow becomes incremental and the program becomes able to parse other custom types derived from the former batch of resolved types, and so on. Thus, types with simple content are the first to be parsed and complex types with complex content are best left for last, since it is necessary to ensure that the types of all of their nested elements are already resolved.

As such, the application starts by looping through the simple content queue recursively, maintaining an array with the names of the finalized types that it updates on every iteration with the newly-parsed types. After clearing the first queue, it repeats the same process for the other one, until it ultimately resolves all custom types.

SIMPLE TYPE DERIVATION ALGORITHM

In order to have the ability to create new custom types by deriving base types, it is necessary to keep track of every type's constrainments, both on their attributes (complex types) and content - simple content (simple types and complex types with simple content) or particles (complex types with complex content).

Let's start at the base of the type dependency hierarchy. For simple types, there are several kinds of derivation:

- **by restriction** - to restrict the lexical space of another type's content, via constraining facets;

- **by list** - to indicate that values of the new type must be whitespace-separated lists of items of the base type. After deriving by list, subsequent constraining facets apply on the list itself and not on the base type;
- **by union** - to indicate that values of the new type may be of any of the types included in the union.

The most relevant of these is **derivation by restriction**, due to its usage of **constraining facets**. When deriving types, their restrictions are cumulative, i.e. type A that derives from type B has both type A's constraining facets and the new ones established in its derivation. As such, in order to be able to derive new simple types from built-in types, the parser requires a structure in memory that maps all of the built-in types to their respective constraining facets, accumulated along this derivation chart:

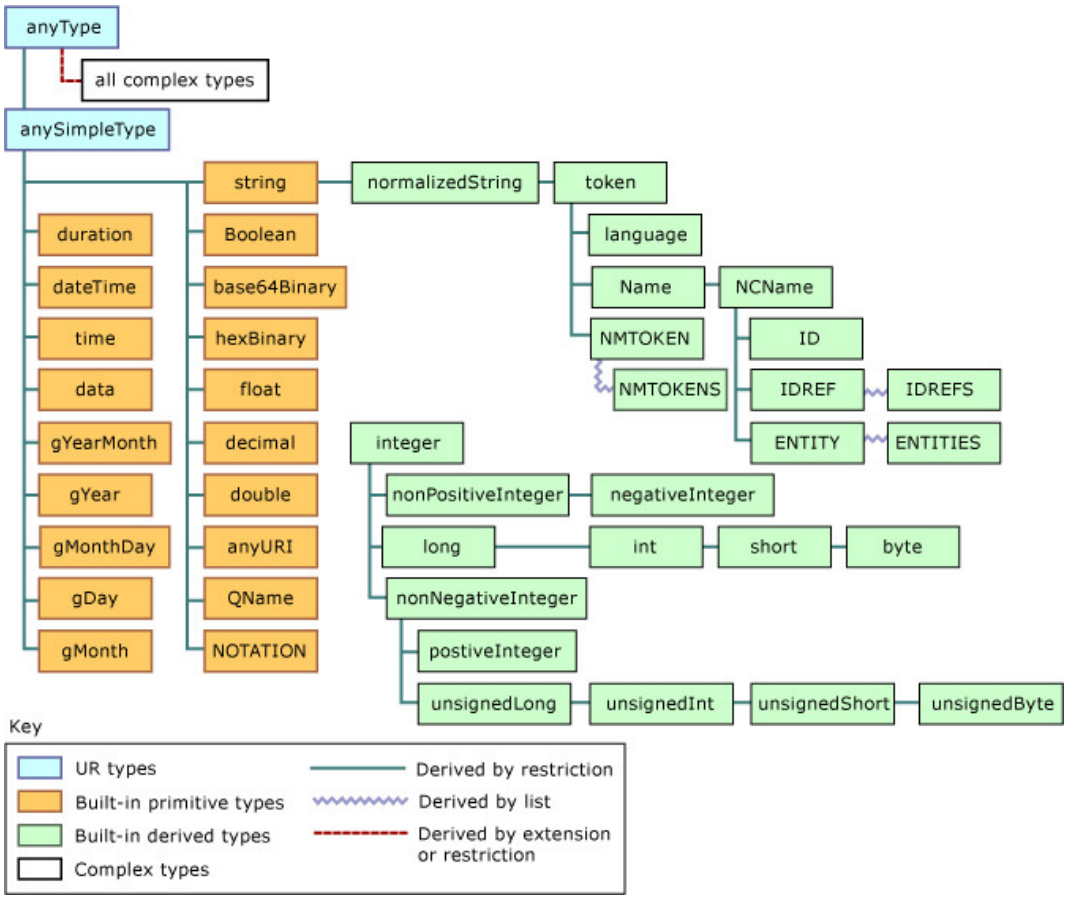


Figure 36: Primitive data types of the *XML* format.

Eric Vlist's book *RELAX NG* (Vlist, 2004) possesses a great reference to all the data types defined by the *W3C XML* Schema, available online here, which actually shows each type's *XSD* declaration and the constraining facets imposed on their bases.

With the aid of this reference to know each type's new constraining facets, *DataGen From Schemas* is able to create these built-in *XSD* types in memory: it starts by defining the primitive types (in orange) and then creating the derived types (in green) in their respective order, e.g. normalizedString from string, token from normalizedString, etc. until reaching the final ones.

With this infrastructure, the solution is then able to create custom types by deriving their base type in the same way. It does this by implementing an algorithm that compares each new contraining facet to the base type's, checking if it is allowed in the type in question, and then incorporating it in a reasonable way, similar to the *JSON* schema extender (4.1.6): if the base type does not possess the new constraining facet, it is added to its content, otherwise the program compares both values of the facet's instances and evaluates what to do with them. For example, in the case of the facet enumeration, they are concatenated, while in the case of maxExclusive, the lower value prevails, as it already implies the other one.

COMPLEX TYPE DERIVATION ALGORITHM

As for complex types, there are two different kinds of derivation:

- **by extension** - used to extend other types with attributes (both simple and complex content) and particles (complex content), i.e. add new content to the base specification;
- **by restriction** - used to restrict other types' contents, namely the lexical space of the content with constraining facets (simple content), or the particles and attributes (complex content).

The derivation of complex types is pretty straightforward, since it basically translates to adding or removing attributes/particles from the base type's structure, which is very easy due to the way in which the parser organizes the element's information (4.2.2). The most important aspect of this operation is the semantic validation of the extension/restriction in question, which must adhere to *XML* Schema's principles of constraints on particle schema components.

4.2.3 *DSL Model Creation*

As was explained in 2.2.1, *DataGen* generates datasets in an intermediate *JSON* object, which is not ideal for representing *XML* data. As such, it was necessary to create a custom key nomenclature to annotate the abnormal cases in the *DSL* model, in order to circumvent this problem and be able to generate all required data correctly.

Custom Key Nomenclature

This section exposes all of the circumstances where custom keys are required, in order to annotate special cases of the *DSL* model that need to be post-processed after generating the dataset, i.e. cases where the *DSL* syntax or intermediate *JSON* object are not enough to depict the *XML* information as is. All of these flags are prefixed with ***DFXS*** (*DataGen From XML Schemas*), in order to be easily identifiable.

Later on, once the final dataset has been generated and is stored in a *JavaScript* object, all that is left to do is to translate it to the intended output format. This process is carried out by *DataGen*'s translators to either language, which were already present in the application's original build. However, these programs had to be greatly expanded, in order to be able to parse these custom keys and modify the dataset according to their meaning, cleaning the structure and creating a final, complete dataset. As such, this section will also expose how these programs post-process the *DSL*'s custom keys and what effect they have on the generated data.

NAME NORMALIZATION

As mentioned at the end of section 3.3, *XML* keys support the characters `.` and `-`, which *JSON* keys do not. As such, any occurrences of these characters need to be replaced with a special flag in order to preserve the original name (encoded) of the element along the generation process. This is applicable to the attribute `name` of elements of both asset types `attribute` and `element`: if their name has any of these characters, then it is prefixed with `DFXS_NORMALIZED_` and dots are replaced with `__DOT__`, while hyphens are replaced with `__HYPHEN__`, for example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="zoo-animals">
    <xs:complexType>
      <xs:all>
        <xs:element name="sea.lion" type="xs:string"/>
        <xs:element name="african-buffalo" type="xs:string"/>
        <xs:element name="komodo.dragon" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  DFXS_NORMALIZED_ELEM__zoo__HYPHEN__animals: {
    at_least(3) {
      DFXS_NORMALIZED_ELEM__african__HYPHEN__buffalo: '{{stringOfSize(5,50)}}',
      DFXS_NORMALIZED_ELEM__komodo__DOT__dragon: '{{stringOfSize(5,50)}}',
      DFXS_NORMALIZED_ELEM__sea__DOT__lion: '{{stringOfSize(5,50)}}'
    }
  }
}
```

Figure 37: *DSL* model with normalized keys produced from the *XML* schema.

In the end, *DataGen*'s translators remove the prefix and either change both flags back to their original characters (*XML*) or replace them with an underscore (*JSON*):

```
{
  "zoo_animals": {
    "african_buffalo": "Kitra",
    "komodo_dragon": "Lincoln",
    "sea_lion": "Morgan"
  }
}
```

```
<?xml version="1.0"?>
<zoo-animals>
  <african-buffalo>Kitra</african-buffalo>
  <komodo.dragon>Lincoln</komodo.dragon>
  <sea.lion>Morgan</sea.lion>
</zoo-animals>
```

Figure 38: Example of instances produced from the previous schema.

ATTRIBUTES

JSON does not support attributes, so these must also be stored in properties of their respective element, along with the remaining content properties. To distinguish attribute properties from others, the program prefixes their name with DFXS_ATTR__ in the *DSL* model.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="text">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="category">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="message"/>
                <xs:enumeration value="email"/>
                <xs:enumeration value="letter"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="purpose" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  text: {
    DFXS_ATTR__category: '{{random("message","email","letter")}}',
    DFXS_ATTR__purpose: '{{stringOfSize(10,50)}}',
    DFXS_SIMPLE_CONTENT: '{{stringOfSize(10,50)}}'
  }
}
```

Figure 39: *DSL* model of an element with simple content and attributes.

In post-processing, the translator to *XML* removes the prefix and concatenates the attribute inside the element's opening tag, writing all attributes inline with the element's name, while its *JSON* counterpart replaces the prefix with attr_, for a cleaner way of differenciating attribute properties, preserving the property in the element's corresponding *JSON* object.

Figure 40: Example of instances produced from the previous schema.

COMPLEX TYPES WITH SIMPLE CONTENT

Simple type elements are represented in the *DSL* model through a simple key/value property, where the key is the element's name and the value is its content's corresponding *DSL* string. However, elements of complex type with simple content can also possess attributes, so in these cases the value must be an object with several properties: attribute properties were explained above, but the content's *DSL* string now also requires a key, to fit into the structure. For this purpose, the program uses the key DFXS_SIMPLE_CONTENT, which can be observed in figure 39.

Ultimately, the translator to *XML* removes this key and pastes its value in the dataset string, between the element's opening and closing tags, while its *JSON* equivalent renames the key to value, indicating that it's the element's content, and leaves the property in its element's corresponding *JSON* object, along with its attribute properties (figure 40).

MIXED CONTENT

If an element has mixed content, the application must annotate its *DSL* string with a special property, in order for it to know that it must generate filler text between its nested elements/properties in post-processing. It is possible to restrict the filler text's content in the schema, so there are two possible keys for this situation:

- DFXS_MIXED_DEFAULT: this key indicates that the element's text can be any type of string and has a filler value (**true**);
- DFXS_MIXED_RESTRICTED: this key indicates that the element's text has been restricted and its value is the corresponding *DSL* string to generate it.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="letter">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="orderid" type="xs:positiveInteger"/>
        <xs:element name="shipdate" type="xs:date"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  letter: {
    DFXS_MIXED_DEFAULT: true,
    DFXS_FLATTEN__1: [ {
      name: '{{stringOfSize(10,50)}}',
      orderid: '{{integer(1, 100001)}}',
      shipdate: '{{xsd_date("1950-01-01")}}'
  }
}
```

Figure 41: *DSL* model of a schema with mixed content.

After the dataset has been generated, *DataGen*'s translator to *XML* removes this flag and produces random text around the element's nested properties, either chunks of *lorem ipsum* (first custom key) or according to its schema restrictions (second custom key). The translator to *JSON* does the same and stores those chunks of text in *JSON* properties with the key text{counter}, since they also need to be in a key/value format.

```json
{
  "letter": {
    "text1": "Dear Mr.",
    "name": "John Smith",
    "text2": ". Your order ",
    "orderid": 1032,
    "text3": " will be shipped on ",
    "shipdate": "2001-07-13",
    "text4": "."
  }
}
```

```xml
<letter>
  Dear Mr. <name>John Smith</name>.
  Your order <orderid>1032</orderid>
  will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

Figure 42: Example of instances produced from the previous schema.

ELEMENTS WITH THE SAME NAME

While in *XML* it is possible to have multiple elements with the same name nested together, which can be specified in the schema, for example, via an element's occurence attributes, *JSON* does not support this feature. Initially, the plan to bypass this was to number every repeated element's occurrences, which was suggested in 2.2.1. However, this would imply having to decide the entire dataset's structure before creating the *DSL* model - e.g. for an element that could occur between 3 and 10 times, its exact number of occurrences would have to be determined *a priori* by the program, which would then write said number of properties of that element to the model. This approach has two big problems:

1. the dataset's structure is pre-programmed into the *DSL* model and not randomly determined from it in runtime, which was the initial intent. With this method, the number of occurrences of any element will be hard-coded into the model, vastly restricting its flexibility and reusability, since only the values of the leaves would be created in runtime;

2. with more complex schemas, this can originate enormous models, which are hard to maintain, read and edit, making them very impractical to use. Furthermore, the requests' response times would skyrocket and models would potentially be too big for response bodies, which could crash the program when trying to serve the model and dataset to the user.

As such, another approach was chosen: using *DataGen*'s repeat function, which specifies that a certain *DSL* chunk must be generated a given number of times, creating an array with the indicated size. The only problem is that this function produces an array and instances of *XML* elements with multiple occurrences should be nested inside the containing element and not an enveloping list.

Therefore, in these cases, the program uses the repeat function to configure a random number of occurrences in the *DSL* model and sets it as the value of a provisory property with the key DFXS_FLATTEN__{counter}, which indicates that the value must be flattened in post-processing. This key also possesses a counter that is incremented each time the repeat function is used, so that multiple properties of this nature don't overwrite each other, if present inside the same containing structure.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="zooAnimals">
    <xs:complexType>
      <xs:sequence maxOccurs="3">
        <xs:element name="elephant" type="xs:string"/>
        <xs:element name="bear" type="xs:string" maxOccurs="2"/>
        <xs:element name="giraffe" type="xs:string"/>
        <xs:element name="lion" type="xs:string" maxOccurs="3"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  zooAnimals: {
    DFXS_FLATTEN__1: [ 'repeat(1,3)': {
      elephant: '{{stringOfSize(10,50)}}',
      DFXS_FLATTEN__2: [ 'repeat(1,2)': {
        bear: '{{stringOfSize(10,50)}}'
      } ],
      giraffe: '{{stringOfSize(10,50)}}',
      DFXS_FLATTEN__2: [ 'repeat(1,2)': {
        lion: '{{stringOfSize(10,50)}}'
      } ]
    } ]
  }
}
```

Figure 43: *DSL* model of a schema with multiple occurrences of elements.

After the dataset is generated, *DataGen*'s translators flatten the array, i.e. extract its values to the root of the containing structure and delete the temporary property, additionally numbering them when translating to *JSON* so that they don't overwrite each other.

```
{
  "zooAnimals": {
    "elephant1": "Aspen",
    "bear1": "Jackson",
    "bear2": "Augustus",
    "giraffe1": "Kijani",
    "lion1": "Desiu",
    "elephant2": "Tongass",
    "bear3": "Rose-Tu",
    "giraffe2": "Cricket",
    "lion4": "Victoria",
    "lion5": "Chendra"
  }
}
```

```
<?xml version="1.0"?>
<zooAnimals>
  <elephant>Aspen</elephant>
  <bear>Jackson</bear>
  <bear>Augustus</bear>
  <giraffe>Kijani</giraffe>
  <lion>Desiu</lion>
  <bear>Rose-Tu</bear>
  <giraffe>Cricket</giraffe>
  <lion>Victoria</lion>
  <lion>Chendra</lion>
</zooAnimals>
```

Figure 44: Example of instances produced from the previous schema.

TRANSPARENT ENVELOPING ELEMENTS

XML Schema has several asset types meant to specify a group of elements through an enveloping element that does not appear itself in the instance: all, choice, and sequence. group does not count since it is simply used to declare and reference top level, resusable elements of the other three asset types.

Well, choice elements allow one and only one of their nested elements to be present in the instance, however their nested elements can be of asset types element, choice, sequence or, indirectly, all (referenced via a group). If a choice possesses a nested all/sequence element and it is selected, all of its nested elements should be present in the final instance.

However, since *DataGen* parses *DSL* models from leaves to root, in this case the nested elements of all/sequence would be shuffled/ordered and only then would the choice select an element, meaning it would also interpret the shuffled/ordered elements as alternatives and not a whole. To prevent this, the application stores the *DSL* string of the element all/sequence inside a temporary object, to basically "bubble-wrap" its nested elements as a whole, and this object is stored in a property with the key DFXS_TEMP__{counter}. The key possesses a counter that is incremented for every new temporary object of this kind that is created, to prevent these properties from overwriting each other, in case there are multiple at the same depth.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:group name="both">
    <xs:all>
      <xs:element name="dog" type="xs:string"/>
      <xs:element name="cat" type="xs:string"/>
    </xs:all>
  </xs:group>

  <xs:element name="pets">
    <xs:complexType>
      <xs:choice>
        <xs:element name="dog" type="xs:string"/>
        <xs:element name="cat" type="xs:string"/>
        <xs:group ref="both"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  pets: {
    DFXS_FLATTEN__1: [ {
      or() {
        dog: '{{stringOfSize(5,50)}}',
        cat: '{{stringOfSize(5,50)}}',
        DFXS_TEMP__2: {
          at_least(2) {
            dog: '{{stringOfSize(5,50)}}',
            cat: '{{stringOfSize(5,50)}}'
          }
        }
      }
    } ]
  }
}
```

Figure 45: *DSL* model with a transparent enveloping element produced from the schema.

In the end, *DataGen*'s translators omit this enveloping property and copy its content to the containing structure:

```
{
  "pets": {
    "cat": "Benny",
    "dog": "Buster"
  }
}
```

```xml
<?xml version="1.0"?>
<pets>
  <cat>Benny</cat>
  <dog>Buster</dog>
</pets>
```

Figure 46: Example of instances produced from the previous schema.

Translation of Elements

With all the custom keys established above, the translation program is finally able to generate a *DSL* model, by iterating the intermediate structure with the schema's data recursively, generating the *DSL* string from leaves to root. It receives only the structure of the top level element that will be instantiated, which possesses all the necessary information for its complete parsing, at this point.

ELEMENTS AND ATTRIBUTES

When translating elements of these asset types, the application checks their attributes before the content, according to a specific order:

- nillable (element-only) - if the element is nillable, its *DSL* string corresponds to an **if**/**else** statement: the if condition has a default probability of 30% (customizable by the user) of being true, in which case the program produces a property with a nil attribute and no additional content; otherwise, it creates a property with the element's regular content:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="middleName" nillable="true"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  name: {
    DFXS_FLATTEN__1: [ {
      firstName: '{{stringOfSize(5,50)}}',
      if (Math.random() < 0.3) {
        middleName: { DFXS_ATTR__nil: true }
      }
      else { middleName: '{{stringOfSize(5,50)}}' },
      lastName: '{{stringOfSize(5,50)}}'
    } ]
  }
}
```

Figure 47: *DSL* model of a schema with a nillable element.

- use (attribute-only) - if this attribute's value is "prohibited", then the program returns an empty string, so that it is not present in the instance;
- fixed - if the element has a fixed value, it simply transcribes said value to the model;
- default - if the element has a default value, its model follows the same structure as with nillable and produces said value only if the condition evaluates to true.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="text" default="This is just a draft.">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="category"
                        type="xs:string" fixed="draft"/>
          <xs:attribute name="purpose"
                        type="xs:string" use="prohibited"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  DFXS_TEMP__1: {
    if (Math.random() < 0.6) {
      text: "This is just a draft."
    }
    else {
      text: {
        DFXS_ATTR__category: "draft",
        DFXS_SIMPLE_CONTENT: '{{stringOfSize(5,50)}}'
      }
    }
  }
}
```

Figure 48: *DSL* model of a schema that uses the attributes use, fixed, and default.

Furthermore, if the element's content is specified with a *DataGen* interpolation function via custom comment, as will be described in 4.3, the program declares the property's value as the result of such function:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string">
          <!--datagen:firstName-->
        </xs:element>
        <xs:element name="lastName" type="xs:string">
          <!--datagen:surname-->
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  name: {
    firstName: '{{firstName()}}',
    lastName: '{{surname()}}'
  }
}
```

Figure 49: *DSL* model of a schema with interpolation functions.

Finally, if the element's *DSL* string still has not been finished at this point, then it is necessary to translate its content, by parsing either the attribute type or, in its absence, the nested simpleType/complexType element.

TYPES WITH SIMPLE CONTENT

DataGen's original grammar was equipped with a lot of new interpolation functions for generating relevant types of data, mostly related with built-in *XSD* simple types, such as hexBinary, xsd_duration, and xsd_gDay, among many others, which are all thoroughly documented in the website. Thus, the content of these types is usually easy to specify in the *DSL*, with most types having direct translation to interpolation functions, along with their restrictions.

However, there are a few exceptions:

- if the type is restricted with one or multiple enumeration facets, its value is simply declared as a random choice between the alternatives, with *DataGen*'s random function:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="car">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Audi"/>
        <xs:enumeration value="Golf"/>
        <xs:enumeration value="BMW"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  car: '{{random("Audi","Golf","BMW")}}'
}
```

Figure 50: *DSL* model of a schema with enumeration facets.

- if it is restricted with a pattern facet, the program uses *DataGen*'s function with the same name to generate a value from the regular expression:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="car">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="Audi|Golf|BMW"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  car: '{{pattern("^(Audi|Golf|BMW)$")}}'
}
```

Figure 51: *DSL* model of a schema with a pattern facet.

- for types ID and IDREF, their value is set as plain strings '{DFXS_ID}' or '{DFXS_IDREF}', respectively, and resolved by *DataGen*'s translator programs after all data is generated. This is so that the application doesn't create any repeated ids or invalid references, since it is very difficult or even impossible to make sure of that with just the syntax of the *DSL*. In post-processing, the translators go through the entire dataset from beggining to end, firstly replacing each ID value with an incrementing id (id1, id2, ...) and then replacing each IDREF value with one of the previous ids (randomly chosen), thus ensuring that the property references a valid id of the schema:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="graph">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="node" minOccurs="3" maxOccurs="3">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="id" type="xs:ID"/>
                <xs:attribute name="neighbor" type="xs:IDREF"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  graph: {
    DFXS_FLATTEN__1: [ {
      DFXS_FLATTEN__2: [ 'repeat(3)': {
        node: {
          DFXS_ATTR__id: '{DFXS_ID}',
          DFXS_ATTR__neighbor: '{DFXS_IDREF}',
          DFXS_SIMPLE_CONTENT: '{{stringOfSize(5,50)}}'
        }
      } ]
    } ]
  }
}
```

```xml
<?xml version="1.0"?>
<graph>
  <node id="id1" neighbor="id2">Ad mollit ad sunt labore tempor in nos</node>
  <node id="id2" neighbor="id3">Enim minim Lor</node>
  <node id="id3" neighbor="id1">Aliquip fugiat </node>
</graph>
```

Figure 52: *DSL* model of a schema with ids and id references, and example *XML* instance.

- union - the *DSL* translation of an union is a random choice between the *DSL* strings of all the possible types, using *DataGen*'s random function:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="jeans_size">
    <xs:simpleType>
      <xs:union memberTypes="sizeByNo sizeByString"/>
    </xs:simpleType>
  </xs:element>
  <xs:simpleType name="sizeByNo">
    <xs:restriction base="xs:positiveInteger">
      <xs:maxInclusive value="42"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="sizeByString">
    <xs:restriction base="xs:string">
      <xs:pattern value="small|medium|large"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  jeans_size: gen => {
    return gen.random(
             gen.integer(1,42),
             gen.pattern("^(small|medium|large)$")
           )
  }
}
```

Figure 53: *DSL* model of a schema with an union element.

- list - the program creates a *JavaScript* function which generates lists of variable size at runtime, within reason. If the list's values' content can only have one type, then the function implements a loop with a randomly determined size, where each iteration generates a random value from its content's *DSL* translation:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="int_values" type="value_list"/>
  <xs:simpleType name="value_list">
    <xs:list itemType="xs:int"/>
  </xs:simpleType>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  int_values: gen => {
    let elems = []
    for (let i = 0; i < Math.floor(Math.random()*(5-2))+2; i++)
      elems.push(gen.integer(-2147483648, 2147483647))
    return elems.join(" ")
  }
}
```

Figure 54: *DSL* model of a schema with a single-type list element.

Otherwise, if the content of each value is an union of several types, it does basically the same, additionally choosing one of the types randomly in each iteration. In the end, the function concatenates all values in a string, separated by spaces:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="not_null_integers">
    <xs:simpleType>
      <xs:list itemType="list_value_types"/>
    </xs:simpleType>
  </xs:element>
  <xs:simpleType name="list_value_types">
    <xs:union memberTypes="xs:negativeInteger
                           xs:positiveInteger"/>
  </xs:simpleType>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  not_null_integers: gen => {
    let elems = []
    for (let i = 0; i < Math.floor(Math.random()*(5-2))+2; i++) {
      let values = [gen.integer(-100001, -1), gen.integer(1, 100001)]
      let next = values[Math.floor(Math.random()*(values.length))]
      elems.push(next)
    }
    return elems.join(" ")
  }
}
```

Figure 55: *DSL* model of a schema with a multi-type list element.

TYPES WITH COMPLEX CONTENT

In order to translate elements with complex typing, the program must be able to create *DSL* strings for all, choice and sequence elements, as well as attributes and mixed content. There is an order to these components in the *DSL* string: mixed content must be the first thing to be discriminated in the element's *DSL* string, with a property as described in 4.2.3; next, the attributes, if there are any - their *DSL* strings must appear before the content's, for easier parsing and organization of the instance; finally, the content's specification, which is built from translating the nested particles.

- all - as predicted in 2.2.4, elements of this asset type are translated with *DataGen*'s at_least function, exactly as exemplified. Furthermore, if the element has a minimum number of occurrences of zero, its model becomes an **if**/**else** statement similar to the one described in 4.2.3, so that users can control the odds of the element occurring, otherwise it would not do so in a lot of generation attempts, which is not as interesting:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="household_members">
    <xs:complexType>
      <xs:all minOccurs="0">
        <xs:element name="mother" type="xs:string"/>
        <xs:element name="father" type="xs:string"/>
        <xs:element name="son" type="xs:string"/>
        <xs:element name="daughter" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  household_members: {
    if (Math.random() < 0.3) {
      missing(100) {empty: true}
    }
    else {
      at_least(4) {
        mother: '{{stringOfSize(5,50)}}',
        father: '{{stringOfSize(5,50)}}',
        daughter: '{{stringOfSize(5,50)}}',
        son: '{{stringOfSize(5,50)}}'
      }
    }
  }
}
```

Figure 56: *DSL* model of a schema with an all element.

- choice and sequence - an element of these asset types needs to always be specified with the custom key DFXS_FLATTEN, even if the particle only occurs a single time, because there may be other elements with the same name as some of its nested elements, in the containing structure, which would result in properties being overwritten and data loss. The custom key prevents this and does not need to be followed by the repeat function if the particle is unitary, as that would be redundant (it is enough the place the particle's content inside an array, to be flattened later):

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="zooAnimals">
    <xs:complexType>
      <xs:sequence maxOccurs="3">
        <xs:element name="elephant" type="xs:string"/>
        <xs:element name="bear" type="xs:string"/>
        <xs:choice>
          <xs:element name="giraffe" type="xs:string"/>
          <xs:element name="lion" type="xs:string"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<!LANGUAGE pt>
{
  zooAnimals: {
    DFXS_FLATTEN__1: [ 'repeat(1,3)': {
      elephant: '{{stringOfSize(5,50)}}',
      bear: '{{stringOfSize(5,50)}}',
      DFXS_FLATTEN__2: [ {
        or() {
          giraffe: '{{stringOfSize(5,50)}}',
          lion: '{{stringOfSize(5,50)}}'
        }
      } ]
    } ]
  }
}
```

Figure 57: *DSL* model of a schema with choice and sequence elements.

4.3 INTEGRATION OF INTERPOLATION FUNCTIONS ON SCHEMAS

DataGen From Schemas produces a *DSL* model along its pipeline, which translates the schema's requisites for the intended dataset's structure and semantics, as an intermediate step of the data generation process. This model is also made available to the user in the application, along with the end result, in order to allow further customizability of the resulting instance in *DataGen*. As such, the user is able to take the produced model and edit it directly in the application's prior version, which can be useful to further detail the content of specific fields or to tweak the specification in ways that require a lot more effort in the schema's dialect. Let's look at the following example in *JSON* Schema:

```
{
  "$id": "https://datagen.di.uminho.pt/json-schemas/person",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "surname": { "type": "string" },
    "birth_date": { "type": "string" },
    "nationality": { "type": "string" },
    "profession": { "type": "string" }
  },
  "additionalProperties": false,
  "required": [ "name", "surname", "birth_date", "nationality", "profession" ]
}
```

Figure 58: *JSON* schema of a person.

The above schema represents a **person**, which must have a **name**, **surname**, **date of birth**, **nationality**, and **profession**. Naturally, all of these properties must be strings, however just restricting them to the string type is too loose a specification, since it presents no impositions on each property's semantics. This schema can validate against instances which make no sense realistically, if the properties' string contents do not match their keys. Likewise, when generating data from the schema as it is, without further restrictions, *DataGen From Schemas* will produce random strings for each property, which is not ideal.

4.3.1 *Possible solutions*

The schemata languages do not possess reasonable means to circumvent this problem, since the most viable option would likely be to hard-code several possible values for each property via an enumeration, which is too cumbersome and time-consuming to be a feasible solution.

The user can, however, modify the *DSL* model directly in *DataGen* with **interpolation functions**, editing each property's model to a directive that better fits its purpose. *DataGen* possesses a vast array of support datasets, which contain vast ammounts of information on certain topics, e.g. animals, brands, names, and jobs. With interpolation functions, it is possible to map a field's content to a random element of one of these datasets, which is incredibly valuable in the context of data generation and exactly what is needed to solve the matter at hand. There are also many other interpolation functions of **spontaneous generation**, which produce specific types of values at runtime, namely dates, unique identifiers, coordinates, and times, which are also very useful in this situation. However, this solution is still not ideal, since the user would always need to edit the intermediate *DSL* model manually in *DataGen*, which breaks the program's quick schema-to-dataset intended workflow, requiring an extra ammount of steps and a decentralized approached for the user to obtain the intended instance.

Therefore, the ideal solution would be the ability to specify properties' contents with *DataGen*'s interpolation functions directly in the schema, making the program more versatile and allowing the user to provide all necessary information in one go.

4.3.2 *Chosen Approach*

The objective was to implement this feature without modifying the schemata languages, in order to preserve their original syntax - the solution that came to mind was to utilize **comments** to provide such instructions, since this application completely ignores them when parsing schemas, given that they have no reflection whatsoever on the produced instance. As such, by defining a strict nomenclature for these custom comments, it would be possible to distinguish them from normal ones and enrich the schema with additional context-specific information for the program, allowing it to produce datasets compliant with the schema that actually reflect the user's intended semantics.

This feature works under the following rules:

- the utilized interpolation function substitutes the content of the element originally established in the schema, in the intermediate *DSL* model. This creates a reusable model that can generate different instances, all according to the user's instructions, instead of only strictly compliant datasets;
- this functionality can only be used with elements/properties that have **elementary content** (numeric types, strings, and booleans). For a field with complex content, it is always possible to customize its elementary local properties individually;
- *DataGen*'s **support datasets** possess data both in portuguese and english. Users should indicate their preference in the program's settings;

- in accordance with point 4.3.2, the only disallowed interpolation functions are random , political_party , and pt_entity, since these (can) generate complex structures, i.e. objects and arrays. Every other *DataGen* interpolation function is permitted, be it a spontaneous generation function or one with dataset support.

XML Schema

In order to implement this mechanism for *XML* Schema, it was necessary to adapt the idea to the language's intrincacies, analyzing which types of elements would be apt for these instructions, where the comments should be integrated and how they would relate with the elements' attributes. Finally, the following convention was established:

- it is only possible to personalize the content of element and attribute elements. All other types of *XSD* elements are used to structure and organize the schema and only these two act as its leaves, holding simple-typed information;
- the element's base-type must be a simple type (simpleType). It is not possible to use interpolation functions with complex type elements (complexType), for the same reason that was stated in point 4.3.2;
- the interpolation function substitutes the element's content (which is defined with the attribute type or a nested simpleType element), but it does not override the its attributes - attributes such as nillable , fixed, default, and use will still be considered during the generation of the model;
- the intended function must be given in a comment **nested inside the element**, before any other nested element, and it must be written in the format <!--datagen:[function_name][arguments]-->, for example:
  - <!--datagen:firstName()-->;
  - <!--datagen:time("hh:mm:ss", 12, false)-->.
- in case of an interpolation function without arguments, it is enough to indicate only the function's name (it is still possible to write empty arguments, only unnecessary), e.g.:
  - <!--datagen:animal-->;
  - <!--datagen:firstName-->.
- otherwise, the function must be written exactly as it would be in *DataGen*, e.g.:
  - <!--datagen:pt_county("district", "Braga")-->;
  - <!--datagen:time("hh:mm:ss", 12, false)-->.

Thus, the previous schema can be rewritten as seen below, in order to map each element's content to adequate interpolation functions, resulting in coherent instances:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person"/>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string">
        <!--datagen:firstName-->
      </xs:element>
      <xs:element name="surname" type="xs:string">
        <!--datagen:surname-->
      </xs:element>
      <xs:element name="birth_date" type="xs:string">
        <!--datagen:date("01-06-1960","15-08-1985","YYYY-MM-DD")-->
      </xs:element>
      <xs:element name="nationality" type="xs:string">
        <!--datagen:nationality-->
      </xs:element>
      <xs:element name="profession" type="xs:string">
        <!--datagen:profession-->
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```xml
<?xml version="1.0"?>
<person>
  <name>Madison-Rose</name>
  <surname>Vorderbruggen</surname>
  <birth_date>1974-02-18</birth_date>
  <nationality>Chinese</nationality>
  <profession>Hat blocking machine operator</profession>
</person>
```

Figure 59: Modified *XML* schema and resulting dataset.

JSON Schema

As for *JSON* Schema, another ordeal emerged: *JSON* does not have native support for comments in its key/value structure, as it is a **data-only format**. As such, for comments to be included in the schema or instance, they must be data too.

The usual approach is to have a designated data element named _comment (or some other similar keyword) that should be ignored by the application that uses the *JSON* data. As such, since it is necessary to introduce a custom keyword in the syntax to support comments, it is better to simply define a _datagen data element in the case of *DataGen From Schemas*, given that the program has no use for any other kind of comments, hence providing a more intuitive keyword to indicate interpolation functions in a more straightforward manner. After deciding this, the following convention was established for this type of schemata:

- it is only possible to customize the content of (sub)schemas of type string, boolean, integer, and number, i.e. elementary types, such as with *XML* Schema - the data type produced by the interpolation function must naturally match the type of schema where it is used;
- the function overrides the schema's type-specific content, but generic keywords such as const, enum, and default remain in effect;

- the intended function must be given as the **string value of the keyword** _datagen, in the schema in question, and must follow the format "_datagen": "[function_name][arguments]", for example:
  - "_datagen": "firstName()";
  - "_datagen": "time('hh:mm:ss', 12, false)".
- if the interpolation function takes no arguments, the user can simply provide its name, e.g.:
  - "_datagen": "animal";
  - "_datagen": "firstName".
- otherwise, the function must be written as it usually would in *DataGen*, except the string arguments must be encased by apostrophes instead of quotation marks, since these are already in use around the keyword's value:
  - "pt_county('district', 'Braga')";
  - "time('hh:mm:ss', 12, false)".

With this, it becomes possible to embed *DataGen*'s interpolation functions in *JSON* schemas. In the case of the schema in figure 69, the result is:

```
{
  "$id": "https://datagen.di.uminho.pt/json-schemas/person",
  "type": "object",
  "properties": {
    "name": { "type": "string", "_datagen": "firstName" },
    "surname": { "type": "string", "_datagen": "surname" },
    "birth_date": {
        "type": "string",
        "_datagen": "date('01-06-1960', '15-08-1985', 'YYYY-MM-DD')"
    },
    "nationality": { "type": "string", "_datagen": "nationality" },
    "profession": { "type": "string", "_datagen": "job" }
  },
  "additionalProperties": false,
  "required": [ "name", "surname", "birth_date", "nationality", "profession" ]
}
```

```
{
  "name": "Calisse",
  "surname": "Graefenstein",
  "birth_date": "1981-11-01",
  "nationality": "Dutch",
  "profession": "Automobile technician"
}
```

Figure 60: Modified *JSON* schema and resulting dataset.

4.4 API ROUTES

The four routes mentioned in subsection 3.3 are made available by *DataGen From Schemas*, allowing the usage of this application without recourse to its interface, which enables integration in third party programs.

The requests' body structure differs depending on the type of schema in question and will be explained in detail, just as it is in a segment of the application's interface, where users can access this information and learn how to employ the program's services via *HTTP* routes. These routes return a *JSON* object with the **generated dataset**, in the indicated format, and the respective ***DataGen DSL* model**, both generated from the given schema.

If the request's body is invalid for any reason, whether it is not correctly structured or the schema has errors, the application answers with a failure status and an error message explaining the problem.

4.4.1 *XML Schema Routes*

There are two different routes for this type of schema:

- POST **/api/xml_schema/xml** - generates the instance in *XML*;
- POST **/api/xml_schema/json** - generates the instance in *JSON*.

The body of these *HTTP* requests must have (only) the three following properties:

- **schema** - the *XML* schema from which data is to generated, must be given in a string;
- **element** - the root element of the schema that the instance should concern. An *XML* Schema may have more than one root element, but an instance can only have one, where all of its information is nested;
- **settings** - the user's preferences for the data generation process, regarding the options made available by the program for *XML* schemas. These should be sent in an object with the following properties:
  - **datagen_language** - language of the results of *DataGen*'s interpolation functions with dataset support. Must be either "pt" (portuguese) or "en" (english);
  - **recursion** - object with the recursion limits. Must have the following properties:
    - **lower** - lower boundary of recursion. Must be a non-negative integer;
    - **upper** - upper boundary of recursion. Must be a non-negative integer.
  - **unbounded** - maximum number of occurrences of an element with the attribute maxOccurs set to "unbounded". Must be a non-negative integer;
  - **prob_default** - probability of the instance of an element with the attribute default specified having that preset value. Must be a number between 0 and 100;
  - **prob_nil** - probability of the instance of an element with the attribute nillable set to true having the explicit value nil, instead of its normal content. Must be a number between 0 and 100;

– **prob_noAll** - probability of an all element with the attribute minOccurs set to zero not occurring in the instance. Must be a number between 0 and 100.

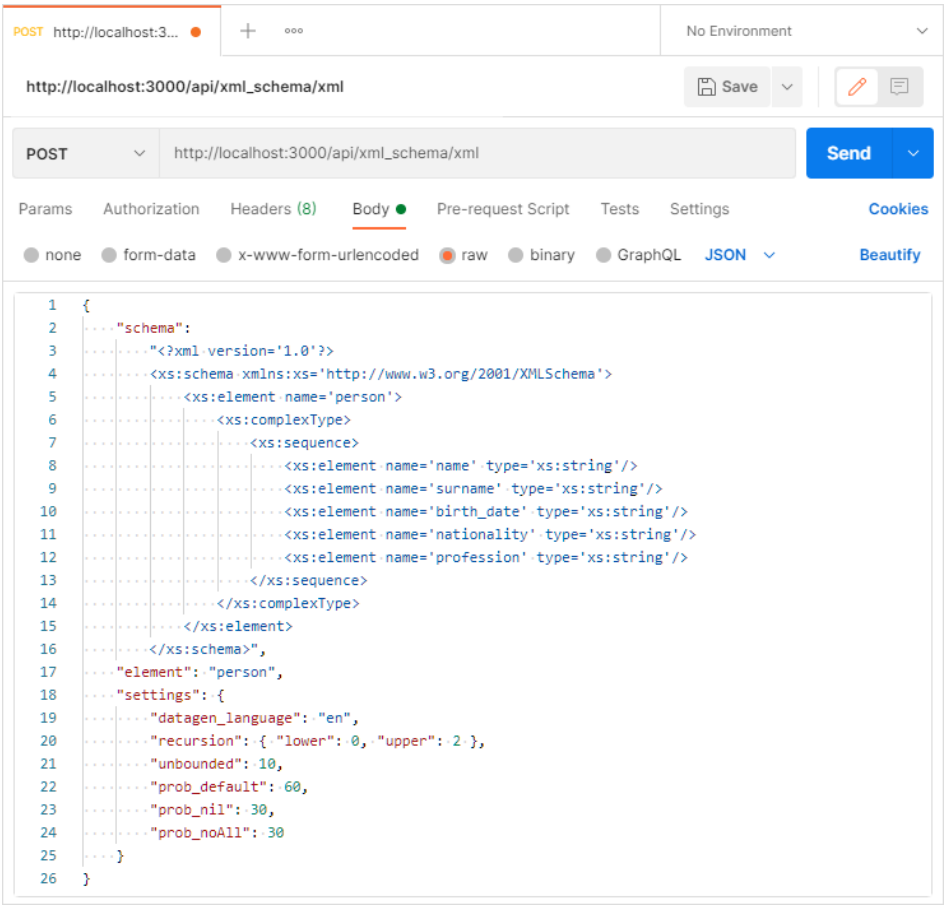


Figure 61: Example request to a *XML* Schema route.

4.4.2 *JSON Schema Routes*

There are two different routes for this type of schema:

- POST **/api/json_schema/json** - generates the instance in *JSON*;
- POST **/api/json_schema/xml** - generates the instance in *XML*.

The body of these *HTTP* requests must have (only) the three following properties:

- **main_schema** - the *JSON* schema from which data is to generated, must be given in a *JSON* object;
- **other_schemas** - an array with the remaining relevant schemas, for cross-schema referencing with the main one. Each of them must also by provided in a *JSON* object;

- **settings** - the user's preferences for the data generation process, regarding the options made available by the program for *JSON* schemas. These should be sent in an object with the following properties:
  - **datagen_language** - language of the results of *DataGen*'s interpolation functions with dataset support. Must be either "pt" (portuguese) or "en" (english);
  - **recursion** - object with the recursion limits. Must have the following properties:
    - **lower** - lower boundary of recursion. Must be a non-negative integer;
    - **upper** - upper boundary of recursion. Must be a non-negative integer.
  - **prob_if** - probability of the schema of a keyword if validating the instance, i.e. the condition being true. Must be a number between 0 and 100;
  - **prob_patternProperty** - probability of generating an instance property from a pattern property, specified in the keyword patternProperty. Must be a number between 0 and 100;
  - **random_props** - possibility of generating additional random properties (while complying with the size designated for the object) if neither of the keywords additionalProperties and unevaluatedProperties are specified. Must be a boolean;
  - **extend_objectProperties** - how to proceed when extending one schema with another, where both have the keyword properties or patternProperties and their values have repeated properties, i.e. what to do when extending those repeated properties' schemas. Must be one of these strings:
    - "extend"/"overwrite" - for every repeated property of these keywords, extend/overwrite its schema of the base keyword with the respective schema of the new keyword. All original properties of the new keyword are also assigned to the base keyword.
  - **extend_schemaProperties** - how to proceed when extending keywords whose value is a subschema (propertyNames, additionalProperties, unevaluatedProperties , items, and unevaluatedItems). Must be one of the following strings:
    - "extend"/"overwrite" - extend/overwrite the schema of the base keyword with the schema of the new one.
  - **extend_prefixItems** - how to proceed when extending the keyword prefixItems. Must be one of the following strings:
    - "extend" - for all schemas at the same index, extend the base keyword's schema with the respective schema from the new keyword. If the new keyword has more elements than the base one, append the extra elements;

* "append" - append the schemas of the new keyword to the base keyword's array;
* "partial_overwrite" - for all schemas at the same index, overwrite the base keyword's schema with the respective schema from the new keyword. If the new keyword has more elements than the base one, append the extra elements;
* "total_overwrite" - the array of schemas of the base keyword is deleted and substituted with the new keyword's.

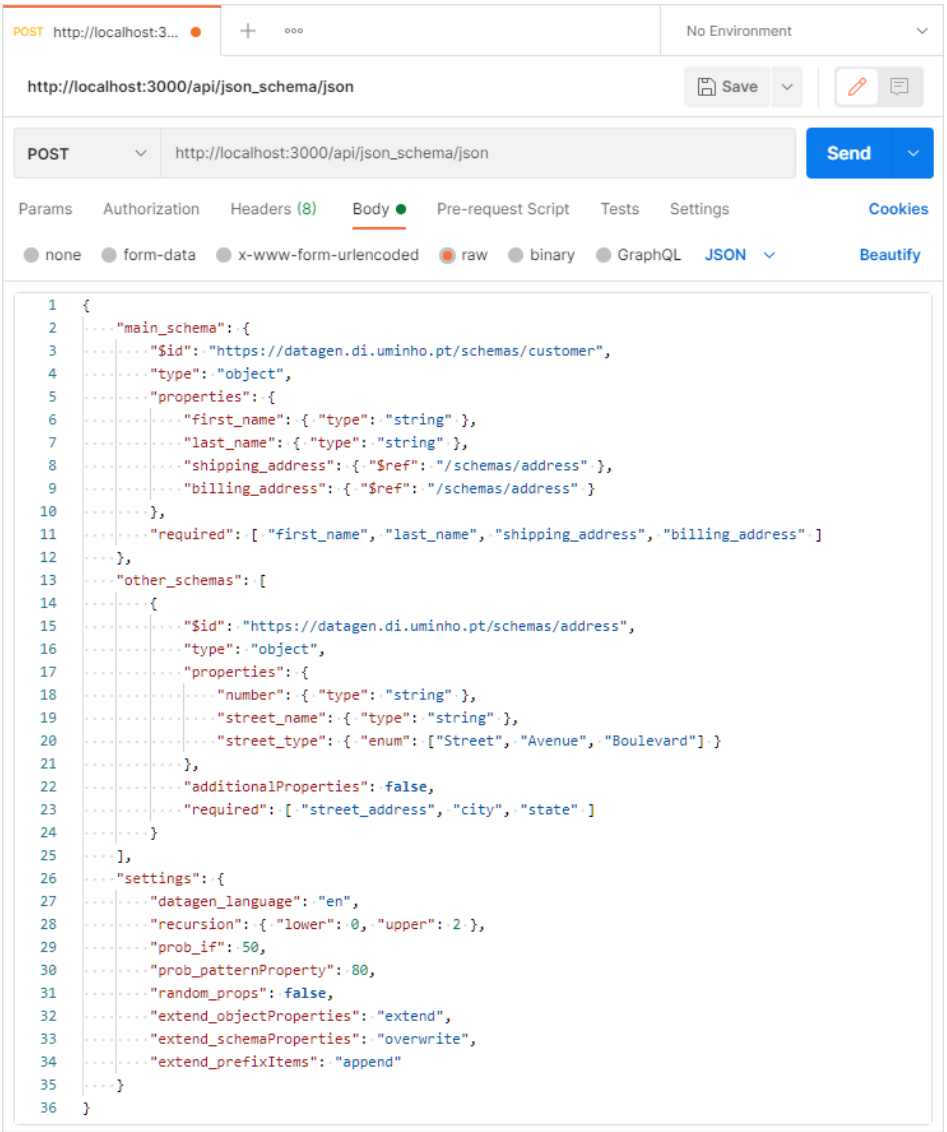


Figure 62: Example request to a *JSON* Schema route.

4.5 GRAPHICAL INTERFACE

The interface of *DataGen From Schemas* was designed in a very similar way to its previous version's, given that the core workflow of both programs is the same and also to maintain a certain familiarity between the two tools, since they can be used complementarily (reason for which it is also in portuguese, just as its predecessor). This is the final look of the application:

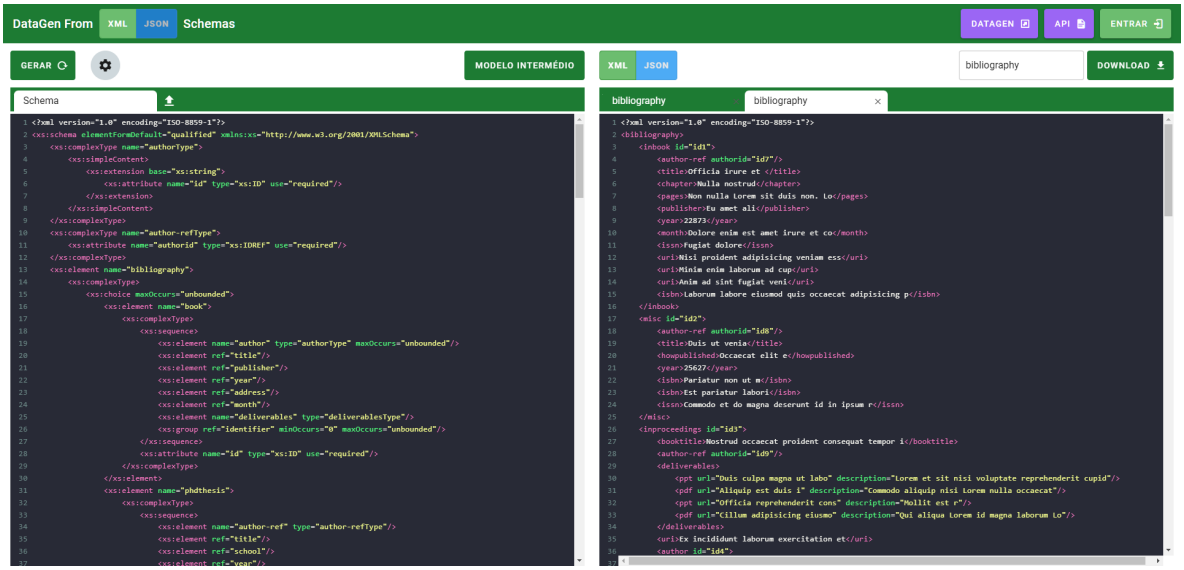


Figure 63: Interface of *DataGen From Schemas*.

The application's name, at the top left corner, possesses a toggle button where users can select the type of schema from which to generate data, *XML* or *JSON*. The interface's color theme changes according to the input schema between green (*XML*) and blue (*JSON*), to signal clearly the type of input schema in use:

Figure 64: Interface of the *JSON* component.

The *JSON* component's interface has two extra features compared to the *XML* component, which are highlighted by the red arrows in the figure above:

1. **a button to add tabs for schemas**: the *JSON* component supports cross-schema referencing, thus its interface allows for the input of multiple schemas. Each individual schema must go into its separate tab and the program automatically detects if the a schema has an id and, if so, changes its tab's name to that;

2. **a tooltip button for information on how to structure complex schemas**: since *DataGen From Schemas* standardizes an *URI* (its own) for the identification of schemas, this information is provided to the users through a modal prompted by this button, otherwise they would have no way of knowing this. This modal also takes the opportunity to show the types of *URI*s (absolute/relative) and references supported by the solution:

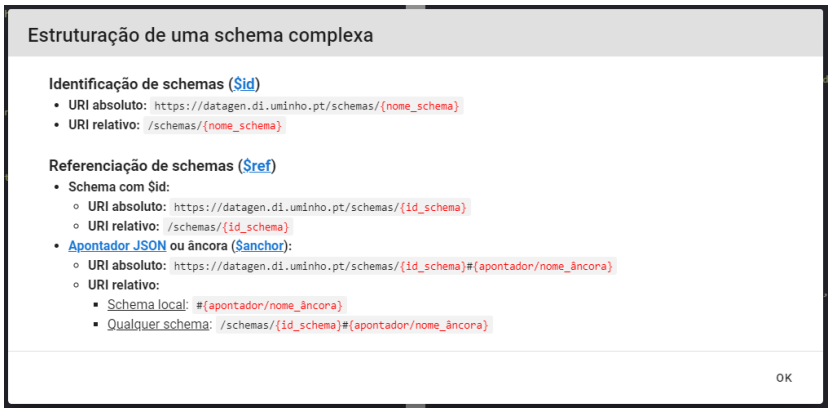


Figure 65: Modal with information on how to structure complex schemas.

Besides this, both components' interfaces share the same set of funcionalities, which will be described with the help of the following screenshot, annotated with numbers in order to easily identify the features in question:

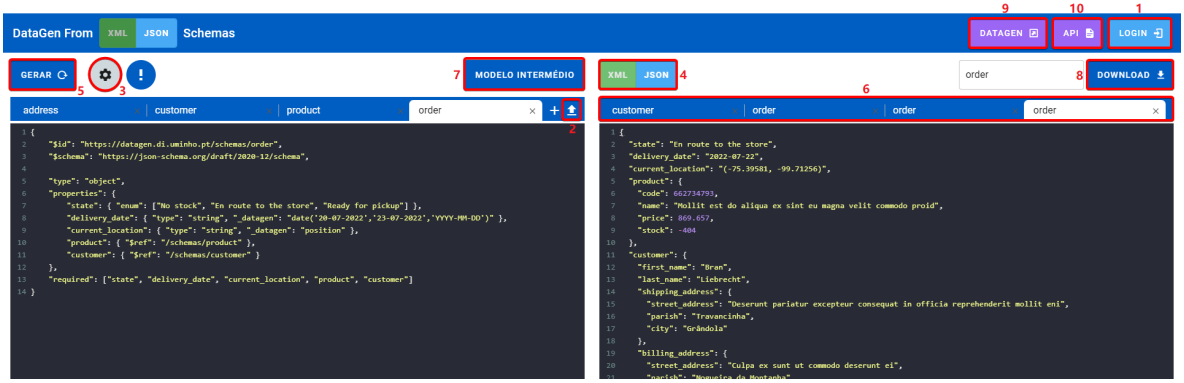


Figure 66: Highlights of the interface's features.

1. **authentication** - the login button opens a modal where users can sign in/up to the website. When logged in, this button is replaced with a logout button and users are able to save *DSL* models directly to their profile, which can be consulted in *DataGen*'s interface;

2. **schema upload** - this button allows users to upload schemas to the program, opening a file explorer for the user to locate the intended file. Uploaded files are verified by extension to ensure that they are in the correct format. In *XML* Schema, the new uploaded schema replaces any previous content of the tab, since the program only allows one schema (no cross-referencing). In *JSON* Schema, unless the current tab is empty, the program will load the new uploaded schema to a new tab;

3. **settings** - each component has its own set of configurable settings for the data generation process, which were already described in 4.4. These options can be personalized in a modal which this button opens:

Figure 67: Settings interface.

4. **output format** - users can choose to generate datasets in either *JSON* or *XML*, regardless of the type of input schema:

5. **dataset generation** - pressing this button triggers the generation process: the application analyzes the schemas, creates an intermediate *DSL* model and generates the final instance in the format intended by the user. With *JSON* Schema, if the user inputs multiple schemas, the program prompts him to select which one should be instanceated. With *XML* Schema, the same happens if the schema has multiple top level elements;

6. **parallel instances** - the application stores each new dataset in a different tab, which means users can generate as many instances as they want and they won't overwrite each other. This way, it is easy to compare different results;

7. **intermediate *DSL* model** - this button opens a modal where users can consult the intermediate *DSL* model generated from their schema, respective to the dataset in the current tab. Besides interacting with the model, users can also execute the following operations on it:
   a) **copy** to the clipboard;
   b) **download**;
   c) **save** to their profile, only if they are logged in.
8. **download** - users can download the selected dataset and quickly specify the file's name in the textbox next to this button;
9. **information on the usage of interpolation functions** - this button prompts a modal where users can find a thorough explanation on how to integrate *DataGen*'s interpolation functions on their schemas, as was described in 4.3. Additionaly, this modal also possesses a button which redirects user to *DataGen*'s website;
10. **information on the *API* routes** - this button prompts a modal with a complete listing and specification of the application's *API* routes, which were also described in 4.4.

As mentioned previously in this dissertation, both components also produce reports on the schema's errors, if there are any, which have the following appearance:

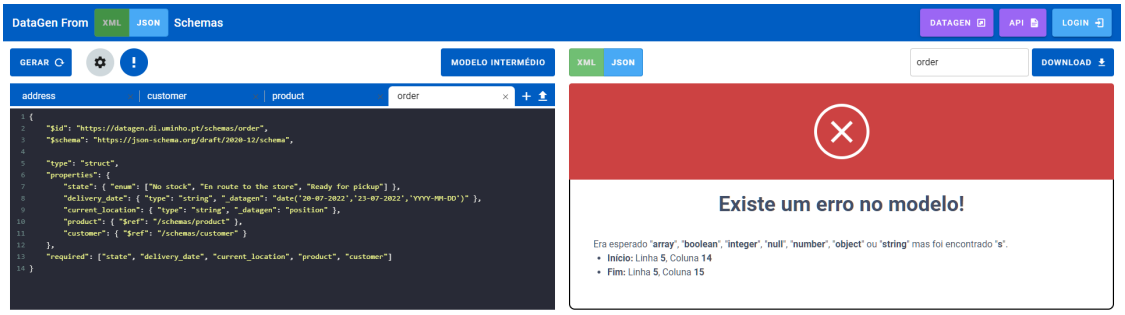


Figure 68: Error report.

Furthermore, if a generation attempt is taking a long time, the application displays a loading circle so that users know that their request is being processed and disables all buttons, to prevent them from submitting additional requests or further inconveniencing the processing of the current one. Each generation attempt has a timeout of 30 seconds, after which the operation is aborted and the user is informed of its expiration:

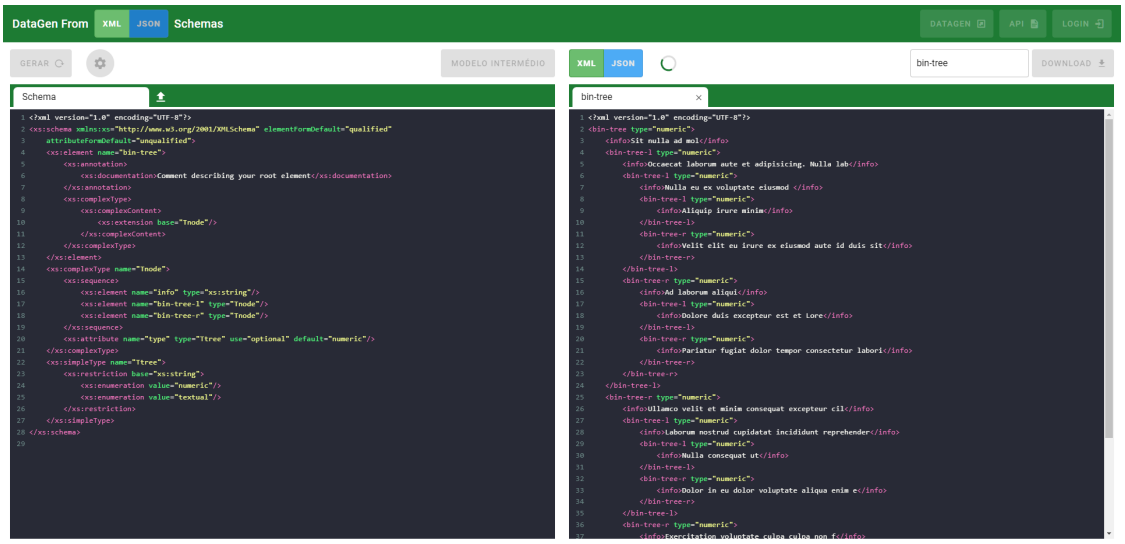


Figure 69: Loading circle on a heavy request (binary tree with 10 levels of recursion).

5

TESTING AND EXAMPLES

The main focus of the testing phase of *DataGen From Schemas* was to gauge the application's usability and sturdiness in terms of the precision of generated instances, in relation to their model, and diversity of use cases, assessing its ability to handle every data specification tool at a schema's disposal - in *JSON* Schema, each relevant keyword and different combinations of them, to check if their semantics coordinated correctly in the program and produced the expected output, mainly with multi-type schemas and schema composition; in *XML* Schema, elements of all the considered asset types, their attributes and possible content, and especially the definition of new, custom types, both simple and complex, as well as different ways of specifying the same schema, on the account of the language's redundancy (mentioned at the start of section 4.2).

These assessments also focused important advanced mechanisms for both schemata languages, namely cross-referencing and bundling in *JSON* Schema, mixed content and definition of custom types in *XML* Schema, and recursion in both.

The tests performed were not very focused on output volume, i.e. ascertaining the product's ability to generate very large datasets from schemas, since this important property was already evaluated and certified with *DataGen*'s first version, which takes over the role of generating data for this new version as well, parsing the *DSL* models it creates from the schemas.

This section will serve to expose some realistic examples of contrived use cases of the program, breaking down the schemas in question and the respective results produced, in order to display the potential and capabilities of *DataGen From Schemas*.

5.1 CROSS-REFERENCING IN JSON SCHEMA

This first example consists of the schema of an order made from the United States of America, which can be observed below:

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/order",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "state": { "enum": [ "No stock", "En route to the store", "Ready for pickup" ] },
    "delivery_date": { "type": "string", "_datagen": "date('20-07-2022','23-07-2022','YYYY-MM-DD')" },
    "current_location": { "type": "string", "_datagen": "position" },
    "product": { "$ref": "/schemas/product" },
    "customer": { "$ref": "/schemas/customer" }
  },
  "required": [ "state", "delivery_date", "current_location", "product", "customer" ]
}
```

Figure 70: Order schema.

An order possesses the following metadata: the package's **state**, which must be one of three alternatives - "No stock", "En route to the store", or "Ready for pickup"; its expected **delivery date**, which must be in the format YYYY-MM-DD, and its **current location**, a set of geographic coordinates, for live tracking. Furthermore, it also stores information relative to the **product** in question and the **customer**, but these details are not defined in the order's schema, but rather in their own, isolated schemas, which the above one points to via external references.

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/product",
  "type": "object",
  "properties": {
    "code": { "type": "integer", "_datagen": "integerOfSize(10)" },
    "name": { "type": "string" },
    "price": { "type": "string", "_datagen": "formattedFloat(0.01, 1500, 2, 1, '0.0,00$')" },
    "stock": { "type": "integer", "minimum": 0, "maximum": 500 }
  },
  "required": [ "code", "name", "price", "stock" ]
}
```

Figure 71: Product schema.

A product's metadata includes its product **code**, a 10-digit reference number, its **name**, the **price** (in dollars) and the available **stock**, which must be a non-negative integer.

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/customer",
  "type": "object",
  "properties": {
    "first_name": { "type": "string", "_datagen": "firstName" },
    "last_name": { "type": "string", "_datagen": "surname" },
    "contact": { "type": "integer", "_datagen": "pattern('^[0-9]{3}-[0-9]{3}-[0-9]{4}$')" },
    "shipping_address": { "$ref": "/schemas/address" },
    "billing_address": { "$ref": "/schemas/address" }
  },
  "required": [ "first_name", "last_name", "contact", "shipping_address", "billing_address" ]
}
```

Figure 72: Customer schema.

As for the customer, its schema specifies a **first** and **last name**, which *DataGen From Schemas* specifies through the corresponding interpolation functions, that allow for the generation of real, used names instead of randomly generated strings, a **phone number**, which must be in the american format XXX-XXX-XXXX, and also the data relative to the **shipping** and **billing addresses**. Once again, there is a separate schema for this metadata, since an address is a set of informations commonly used in different instances, so it makes sense to specify its properties in a separate, reusable schema, in order to avoid duplicating them everywhere and structure schemas in a cleaner way.

An address specifies the **number** of the residency, the **street's name** and its **type**, which must be either a street, an avenue, or a boulevard:

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/address",
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": [ "Street", "Avenue", "Boulevard" ] }
  },
  "required": [ "number", "street_name", "street_type" ]
}
```

Figure 73: Address schema.

DataGen From Schemas analyzes all of these schemas separately, extracting each's data to a different intermediate structure, and resolves the references between the schemas, replacing each reference in the order with its respective structure and centralizing all the necessary information for the creation of the corresponding *DSL* model. The schemas specify the content of properties in very detailed ways, using *DataGen*'s interpolation functions to enrich

the instances with realistic values from support datasets which are otherwise impossible to recreate using only the language's native features.

Generated instances look like the following, in either format, and it is possible to observe that the order's structure now has all of the properties explained above, which the program fetched from the referenced schemas.

```
{
  "state": "En route to the store",
  "delivery_date": "2022-07-22",
  "current_location": "(42.3917229, -8.8273713)",
  "product": {
    "code": 3802482217,
    "name": "Apple Airpods Pro",
    "price": "1.083,54$",
    "stock": 157
  },
  "customer": {
    "first_name": "Yonara",
    "last_name": "Messias",
    "contact": "179-987-9574",
    "shipping_address": {
      "number": 24,
      "street_name": "Pennsylvania",
      "street_type": "Boulevard"
    },
    "billing_address": {
      "number": 16,
      "street_name": "Washington",
      "street_type": "Avenue"
    }
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
  <state>Ready for pickup</state>
  <delivery_date>2022-07-21</delivery_date>
  <current_location>(63.93069, 149.67633)</current_location>
  <product>
    <code>912634912</code>
    <name>Xiaomi Redmi Buds 3 Pro</name>
    <price>441,16$</price>
    <stock>195</stock>
  </product>
  <customer>
    <first_name>Joaquin</first_name>
    <last_name>Gastro</last_name>
    <contact>150-683-0764</contact>
    <shipping_address>
      <number>552</number>
      <street_name>Express</street_name>
      <street_type>Street</street_type>
    </shipping_address>
    <billing_address>
      <number>836</number>
      <street_name>Liberty</street_name>
      <street_type>Street</street_type>
    </billing_address>
  </customer>
</order>
```

Figure 74: Example instances generated from the previous interconnected schemas.

5.2 Recursion in XML Schema

The next example is a **binary tree** schema, which is a recursive structure. In this case, it is possible to observe below that the complex type **Tnode** defines elements with its own typing in its hierarchy, thus creating a possibly infinite loop, which is the reason why *DataGen From Schemas* needs to set an upper boundary for recursion in data structures:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bin-tree">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="Tnode"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="Tnode">
    <xs:sequence>
      <xs:element name="info" type="xs:string"/>
      <xs:element name="bin-tree-l" type="Tnode"/>
      <xs:element name="bin-tree-r" type="Tnode"/>
    </xs:sequence>
    <xs:attribute name="type" type="Ttree" use="optional" default="numeric"/>
  </xs:complexType>
  <xs:simpleType name="Ttree">
    <xs:restriction base="xs:string">
      <xs:enumeration value="numeric"/>
      <xs:enumeration value="textual"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Figure 75: Binary tree schema.

An instance produced by the application from this schema, in this case with two levels of recursion depth, looks like this:

```json
{
  "bin_tree": {
    "attr_type": "numeric",
    "info": "Laborum qui fugiat nulla",
    "bin_tree_l": {
      "attr_type": "numeric",
      "info": "Magna voluptate excepteur nostrud dolor amet. An",
      "bin_tree_l": {
        "attr_type": "numeric",
        "info": "Veniam veniam cupidatat voluptate Lorem"
      },
      "bin_tree_r": {
        "attr_type": "textual",
        "info": "Anim eu fugiat ci"
      }
    },
    "bin_tree_r": {
      "attr_type": "textual",
      "info": "Consectetur occaecat",
      "bin_tree_l": {
        "attr_type": "numeric",
        "info": "Qui ipsum deserunt com"
      },
      "bin_tree_r": {
        "attr_type": "numeric",
        "info": "Pariatur ad culpa minim tempor ex labor"
      }
    }
  }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bin-tree type="numeric">
  <info>Ea exercitat</info>
  <bin-tree-l type="numeric">
    <info>Anim duis eius</info>
    <bin-tree-l type="textual">
      <info>Eu laboris aliqua excepteur incididu</info>
    </bin-tree-l>
    <bin-tree-r type="numeric">
      <info>Commodo ullamco ea aliquip e</info>
    </bin-tree-r>
  </bin-tree-l>
  <bin-tree-r type="numeric">
    <info>Irure voluptat</info>
    <bin-tree-l type="numeric">
      <info>Esse esse</info>
    </bin-tree-l>
    <bin-tree-r type="textual">
      <info>Elit ut Lorem et sunt. Ullam</info>
    </bin-tree-r>
  </bin-tree-r>
</bin-tree>
```

Figure 76: Binary tree instances.

5.3 LARGE DATASETS

To corroborate the application's ability to generate large volumes of data which was mentioned at the beginning of this section of result analysis, there is another *JSON* Schema example of a **graph**, which possesses 100 nodes and 2000 links between them:

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/graph",
  "type": "object",
  "properties": {
    "nodes": {
      "type": "array", "minItems": 100, "maxItems": 100,
      "items": { "$ref": "/schemas/node" }
    },
    "links": {
      "type": "array", "minItems": 2000, "maxItems": 2000,
      "items": { "$ref": "/schemas/link" }
    }
  }
}
```

Figure 77: Graph schema.

Both the node's and the link's structures are specified separately in their respective schemas and referenced externally from the graph's. Each node has an **id**, which is an incrementing integer across all nodes, a **name** and an integer **value**, whereas each link has an **id**, which works as was already described, an **origin** and **destiny nodes** (those nodes' ids), which it links, and a link **weight**, which is an arbitrary numeric value.

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/node",
  "type": "object",
  "properties": {
    "id": { "type": "integer", "_datagen": "index" },
    "name": { "type": "string" },
    "value": { "type": "integer" }
  },
  "additionalProperties": false,
  "required": [ "id", "name", "value" ]
}
```

```
{
  "$id": "https://datagen.di.uminho.pt/schemas/link",
  "type": "object",
  "properties": {
    "id": { "type": "integer", "_datagen": "index" },
    "origin": { "type": "integer", "_datagen": "integer(0, 100)" },
    "destiny": { "type": "integer", "_datagen": "integer(0, 100)" },
    "weight": { "type": "number", "_datagen": "float(1, 300)" }
  },
  "additionalProperties": false,
  "required": [ "id", "origin", "destiny", "weight" ]
}
```

Figure 78: Node and link schemas.

DataGen From Schemas parses these schemas and creates an according *DSL* model, from which it then generates datasets with the amount of units indicated above, that look like the following:

```
{
  "nodes": {
    "0": {
      "id": 1,
      "name": "Est Lorem consequat ",
      "value": 500
    },
    ...
    "99": {
      "id": 99,
      "name": "Eu enim ex veniam ",
      "value": -943
    }
  },
  "links": {
    "0": {
      "id": 1,
      "origin": 97,
      "destiny": 20,
      "weight": 68.349
    },
    ...
    "1999": {
      "id": 223,
      "origin": 73,
      "destiny": 13,
      "weight": 269.98
    }
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<graph>
  <nodes>
    <elem_1>
      <id>1</id>
      <name>Excepteur cupidatat pariatur. </name>
      <value>519</value>
    </elem_1>
     ...
    <elem_100>
      <id>99</id>
      <name>Commodo cillum cupidatat id</name>
      <value>172</value>
    </elem_100>
  </nodes>
  <links>
    <elem_1>
      <id>1</id>
      <origin>83</origin>
      <destiny>66</destiny>
      <weight>36.304</weight>
    </elem_1>
     ...
    <elem_2000>
      <id>1999</id>
      <origin>81</origin>
      <destiny>23</destiny>
      <weight>261.922</weight>
    </elem_2000>
  </links>
</graph>
```

Figure 79: Graph instances.

5.4 BIBTEX

The final showcase is a very complex example of a bibliography management tool, **BibTeX**, used to describe and process lists of references, mostly in conjunction with *LaTeX* documents.

A bibliography is a list with a variable amount of references, which can be citing several different types of source documents: **books**, ***PhD* theses**, **articles**, **proceedings**, **Master's theses** or other **miscellaneous documents**. Each kind of document has its own set of informations that need to be specified for the reference to be valid, which can be seen in the schema, in appendix A.

The program is able to parse this schema and produce wildly different bibliographies according to this structure, such as the following automatically generated example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bibliography>
  <masterthesis id="id1">
    <author-ref authorid="id3"/>
    <title>Ex sunt esse enim nulla rep</title>
    <school>Mollit consectetur nulla velit</school>
    <year>17330</year>
    <month>Ex qui exercitation ut nulla. Culpa dolo</month>
    <uri>Ut officia ad incididunt minim dolore </uri>
    <uri>Non qui cupid</uri>
  </masterthesis>
  <article id="id2">
    <volume>Velit ea adip</volume>
    <journal>Duis veniam sunt laboris sit laborum Lore</journal>
    <deliverables>
      <doc url="Et aliquip ad minim do quis est" description="Commodo mollit r"/>
      <ppt url="Veniam esse fugiat id magna qui dui" description="Amet ad laborum labore"/>
      <pdf url="Ullamco est est aliquip enim ea sit. " description="Ullamco proident amet"/>
      <doc url="Laboris sunt exer" description="Occaecat esse laboris Lorem occaecat fugiat co"/>
      <pdf url="Proident ex ullamco incididunt conseq" description="Incididunt"/>
      <doc url="Fugiat exercitation commodo dolore pariatu" description="Proident amet est aliqua eiu"/>
      <xhtml url="Sunt et enim" description="Minim incididunt ex qui proident ut non adipisic"/>
      <pdf url="Nisi ea consectetur occaecat sunt " description="Esse sunt eu aute sit reprehende"/>
    </deliverables>
    <author id="id3">
            Laborum eu proident id m
    </author>
    <author id="id4">
            Elit eiusmod nisi quis labore est cillum
    </author>
    <author id="id5">
            Amet adipisicing ullamco ullamco culpa te
    </author>
  </article>
  <proceedings id="id6">
    <title>Aliqua non minim vel</title>
    <year>32699</year>
    <editor-ref authorid="jcr"/>
    <editor-ref authorid="prh"/>
    <editor-ref authorid="prh"/>
    <editor-ref authorid="jcr"/>
    <editor-ref authorid="grl"/>
    <editor-ref authorid="prh"/>
    <editor-ref authorid="grl"/>
    <address>Sunt culpa mollit mollit incidid</address>
    <month>Duis officia amet magna et ullamco </month>
    <deliverables>
      <doc url="Et aliquip ad minim do quis est" description="Commodo mollit r"/>
      <ppt url="Veniam esse fugiat id magna qui dui" description="Amet ad laborum labore"/>
      <pdf url="Ullamco est est aliquip enim ea sit. " description="Ullamco proident amet"/>
      <doc url="Laboris sunt exer" description="Occaecat esse laboris Lorem occaecat fugiat co"/>
      <pdf url="Proident ex ullamco incididunt conseq" description="Incididunt"/>
      <doc url="Fugiat exercitation commodo dolore pariatu" description="Proident amet est aliqua eiu"/>
      <xhtml url="Sunt et enim" description="Minim incididunt ex qui proident ut non adipisic"/>
      <pdf url="Nisi ea consectetur occaecat sunt " description="Esse sunt eu aute sit reprehende"/>
    </deliverables>
    <isbn>Amet culpa d</isbn>
  </proceedings>
</bibliography>
```

Figure 80: BibTeX instance.

6

CONCLUSION

The objective of this work was to develop a synthetic dataset generator from *JSON* and *XML* schemata, capable of producing artificial data in both formats from either type of schema. The viability of the solution hinged heavily on generation times and scalability, i.e. its ability to produce possibly large amounts of fake data quickly, while complying while the model's constraints. The project was developed on top of the already existing *DataGen* software (Santos et al., 2021) and achieved all the established goals. The application is available online as an open-source project on Github and also hosted live at `https://datagenfromschemas.di.uminho.pt/`, for public use.

6.1 OUTCOMES

The development of this project as a new version of *DataGen* allowed to build on top of this tool and take advantage of its dataset specification *DSL* and data generation routine, which cover a great part of the new solution's functionality and helped shift the implementation focus to the schema processing and automatic building of equivalent *DSL* models, which were very complicated and extensive processes.

The result is a fast and reliable program that significantly expands the base application's functionality and increases its usability for all users, ditching the necessity to learn the *DSL* and allowing them to instead use schemata in hugely popular and widely utilized languages.

In addition, this software further develops several of its predecessor's features, now supporting the generation of multiple instances in parallel, in contrast to the previous single instance that would be overwriten in following requests, which is great for comparing several results of the same model, as well as the upload of specifications directly to the application, without need for manual writing or copy-pasting. These changes enhance user experience and make this tool easier and more practical to use.

Furthermore, the development philosophy of joint operation between *DataGen* and this new version also resulted in the creation of interesting and useful tools for dataset specification which complement the schema languages with *DataGen*'s tools: the customization of settings

respective to intrincacies of each type of schema with probabilities and design choices, as well as the integration of interpolation functions directly in the schema languages, enabling the definition of concrete, fake content from support datasets, which is impossible in regular schemata and greatly enriches the produced instances. There is also direct compatibility between both programs, since *DataGen From Schemas* provides the intermediate *DSL* model to the user, which can be introduced and edited directly in *DataGen*.

DataGen From Schemas has been vastly tested with complex and exemplary schemas, that can be found in development and production contexts, and proves to be an adequate solution, capable of interpreting schemas correctly and generating representative and sizeable datasets accordingly.

As part of this project, another paper was also written (Cardoso and Ramalho, 2022) and presented at the 2022 edition of the *Symposium on Languages, Applications and Technologies* (*SLATE*), being then officialy published as part of the volume **OASIcs, Volume 104, *SLATE* 2022**. This article covers the development of the *JSON* Schema component, described in chapter 4 of the present dissertation.

6.2 Future Work

Although all of the objectives established for this work were achieved and the final application even accounts for some extra funcionalities that were not initially planned, there is still room for improvement and expansion, which could enhance the current build.

Cross-referencing in XML Schema

The biggest disparity between the program's implementation of *JSON* and *XML* Schema is the limitation to one *XML* schema, which prevents the usage of external references in this language. This would be the priority of further work on *DataGen From Schemas* and would be useful for the structuring and development of more complex *XSD* use cases.

Workarounds on Niche Cases of JSON Schema

As covered in 4.1.4, *JSON* Schema possesses some functionalities that are ill-suited to the solution's workflow of generating instances from schemas, instead of the opposite, for which they were initially designed. This refers mainly to keywords with dynamic semantics, whose value is based on other values of the instance, namely unique and contains. The current implementation of these keywords is limited and does not ensure their correct operability in all use cases, so these features would be a good subject for further development and refining.

Improved Feedback

Another feature that would benefit from additional development time is the application's error feedback system. Presently, the program halts the execution of the pipeline if it finds an error and reports it to the user, discriminating the motive of the request's failure and the error's position in the schema, i.e. the line and column where it starts and ends. Additionaly, in *JSON* Schema, it switches over to the tab of the schema with the error in question.

However, it would be more user-friendly and intuitive to highlight the error directly in the schema, without making the user search for it, which can be especially tedious in lenghty schemas and also because the program's text editor does not have a column counter, as it does for lines.

BIBLIOGRAPHY

General data protection regulation. In *GDPR*, 2018. URL https://gdpr-info.eu/. Accessed: 2021-04-26.

Jason W. Anderson, K. E. Kennedy, Linh B. Ngo, Andre Luckow, and Amy W. Apon. Synthetic data generation for the internet of things. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 171–176, 2014. doi: 10.1109/BigData.2014.7004228.

Hugo André Coelho Cardoso and José Carlos Ramalho. Synthetic Data Generation from JSON Schemas. In João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais, editors, *11th Symposium on Languages, Applications and Technologies (SLATE 2022)*, volume 104 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-245-7. doi: 10.4230/OASIcs.SLATE.2022.5. URL https://drops.dagstuhl.de/opus/volltexte/2022/16751.

Jessamyn Dahmen and Diane Cook. Synsys: A synthetic data generation system for healthcare applications. *Sensors*, 19(5), 2019. ISSN 1424-8220. doi: 10.3390/s19051181. URL https://www.mdpi.com/1424-8220/19/5/1181.

Hadi Keivan Ekbatani, Oriol Pujol, and Santi Segui. Synthetic data generation for deep learning in counting pedestrians. In *ICPRAM*, pages 318–323, 2017.

GAO. Artificial intelligence in health care: Benefits and challenges of machine learning in drug development (staa)-policy briefs & reports-epta network. 2020. URL https://eptanetwork.org/database/policy-briefs-reports/1898-artificial-intelligence-in-health-care-benefits-and-challenges-of-machine-learning-in-drug-development-staa. Accessed: 2021-04-25.

Menno Mostert, Annelien L Bredenoord, Monique Biesaart, and Johannes Delden. Big data in medical research and eu data protection law: Challenges to the consent or anonymise approach. 2016. doi: 10.1038/ejhg.2015.239.

Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, 2016. doi: 10.1109/DSAA.2016.49.

Donald B. Rubin. Statistical disclosure limitation. page 461–468, 1993.

Filipa Alves dos Santos, Hugo André Coelho Cardoso, João da Cunha e Costa, Válter Ferreira Picas Carvalho, and José Carlos Ramalho. DataGen: JSON/XML Dataset Generator. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, volume 94 of *Open Access Series in Informatics (OASIcs)*, pages 6:4–6:9, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-202-0. doi: 10.4230/OASIcs.SLATE.2021.6. URL https://drops.dagstuhl.de/opus/volltexte/2021/14423.

Dianna M Smith, Graham P Clarke, and Kirk Harland. Improving the synthetic data generation process in spatial microsimulation models. *Environment and Planning A: Economy and Space*, 41(5):1251–1268, 2009. doi: 10.1068/a4147. URL https://doi.org/10.1068/a4147.

Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint arXiv:1710.06270*, 2017.

E. Vlist. *RELAX NG*. O'Reilly Series. O'Reilly Media, 2004. ISBN 9780596004217. URL https://books.google.pt/books?id=3pJObV0hrhUC.

A

BIBTEX SCHEMA

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="authorType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="author-refType">
    <xs:attribute name="authorid" type="xs:IDREF" use="required"/>
  </xs:complexType>
  <xs:element name="bibliography">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="author" type="authorType" maxOccurs="unbounded"/>
              <xs:element ref="title"/>
              <xs:element ref="publisher"/>
              <xs:element ref="year"/>
              <xs:element ref="address"/>
              <xs:element ref="month"/>
              <xs:element name="deliverables" type="deliverablesType"/>
              <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:ID" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="phdthesis">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="author-ref" type="author-refType"/>
              <xs:element ref="title"/>
```

```
36              <xs:element ref="school"/>
                <xs:element ref="year"/>
38              <xs:element ref="address"/>
                <xs:element ref="month"/>
40              <xs:element name="deliverables" type="deliverablesType"/>
                <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
42            </xs:sequence>
              <xs:attribute name="id" type="xs:ID" use="required"/>
44          </xs:complexType>
        </xs:element>
46      <xs:element name="article">
          <xs:complexType>
48          <xs:choice maxOccurs="unbounded">
              <xs:element name="author-ref" type="author-refType"/>
50            <xs:element name="author" type="authorType"/>
              <xs:element name="journal" type="xs:string"/>
52            <xs:element ref="month"/>
              <xs:element ref="title"/>
54            <xs:element name="volume" type="xs:string"/>
              <xs:element ref="year"/>
56            <xs:element name="deliverables" type="deliverablesType"/>
              <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
58            <xs:element name="number" type="xs:string"/>
              <xs:element ref="publisher"/>
60          </xs:choice>
            <xs:attribute name="id" type="xs:ID" use="required"/>
62        </xs:complexType>
        </xs:element>
64      <xs:element name="inproceedings">
          <xs:complexType>
66          <xs:choice maxOccurs="unbounded">
              <xs:element name="author-ref" type="author-refType"/>
68            <xs:element name="author" type="authorType"/>
              <xs:element ref="title"/>
70            <xs:element name="booktitle" type="xs:string"/>
              <xs:element ref="year"/>
72            <xs:element ref="address"/>
              <xs:element ref="month"/>
74            <xs:element name="deliverables" type="deliverablesType"/>
              <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
76            <xs:element name="editor" minOccurs="0">
                <xs:complexType>
78                <xs:simpleContent>
                    <xs:extension base="xs:string">
80                    <xs:attribute name="id" type="xs:ID" use="required"/>
                    </xs:extension>
82                </xs:simpleContent>
```

```
            </xs:complexType>
          </xs:element>
        </xs:choice>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="inbook">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="author-ref" type="author-refType"/>
          <xs:element ref="title"/>
          <xs:element name="chapter" type="xs:string"/>
          <xs:element name="pages" type="xs:string"/>
          <xs:element ref="publisher"/>
          <xs:element ref="year"/>
          <xs:element ref="month"/>
          <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="masterthesis">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="author-ref" type="author-refType"/>
          <xs:element ref="title"/>
          <xs:element ref="school"/>
          <xs:element ref="year"/>
          <xs:element ref="month"/>
          <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="misc">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="author-ref" type="author-refType"/>
          <xs:element ref="title"/>
          <xs:element name="howpublished" type="xs:string"/>
          <xs:element ref="year"/>
          <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="proceedings">
```

```
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="title"/>
              <xs:element ref="year"/>
              <xs:element name="editor-ref" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="authorid" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="grl"/>
                        <xs:enumeration value="jcr"/>
                        <xs:enumeration value="prh"/>
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
              <xs:element name="editor" minOccurs="0">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="id" use="required">
                        <xs:simpleType>
                          <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="albie"/>
                            <xs:enumeration value="gva"/>
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element ref="address"/>
              <xs:element ref="month"/>
              <xs:element name="deliverables" type="deliverablesType" minOccurs="0"/>
              <xs:group ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:ID" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="deliverablesType">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="xhtml">
```

```
        <xs:complexType>
          <xs:attribute name="url" type="xs:string" use="required"/>
          <xs:attribute name="description" use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="pdf">
        <xs:complexType>
          <xs:attribute name="url" type="xs:string" use="required"/>
          <xs:attribute name="description" use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="doc">
        <xs:complexType>
          <xs:attribute name="url" type="xs:string" use="required"/>
          <xs:attribute name="description" use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="ppt">
        <xs:complexType>
          <xs:attribute name="url" type="xs:string" use="required"/>
          <xs:attribute name="description" type="xs:string" use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
  <xs:element name="address" type="xs:string"/>
  <xs:element name="month" type="xs:string"/>
  <xs:element name="publisher" type="xs:string"/>
  <xs:element name="school" type="xs:string"/>
  <xs:element name="title" type="xs:string"/>
  <xs:element name="year" type="xs:short"/>
  <xs:group name="identifier">
    <xs:choice>
      <xs:element name="isbn" type="xs:string"/>
      <xs:element name="issn" type="xs:string"/>
      <xs:element name="uri" type="xs:string"/>
    </xs:choice>
  </xs:group>
</xs:schema>
```

B

JSON SCHEMA GRAMMAR (CODE OMITTED)

```
Dialect = ws false ws / ws schema:schema_object ws

begin_array     = ws "[" ws
begin_object    = ws "{" ws
end_array       = ws "]" ws
end_object      = ws "}" ws
name_separator  = ws ":" ws
value_separator = ws "," ws

ws "whitespace" = [ \t\n\r]*

value = boolean / null / object / array / number / string
boolean = false / true

false = "false"
null  = "null"
true  = "true"

// ----- Keywords -----
keyword = datagen_keyword / generic_keyword / string_keyword / number_keyword / object_keyword /
     array_keyword / media_keyword / schemaComposition_keyword / conditionalSubschemas_keyword /
     structuring_keyword

// ---------- Keywords generic ----------
generic_keyword = kw_type / kw_enum / kw_const / kw_default / annotation_keyword

kw_type = QM key:"type" QM name_separator value:type_value
type_value = t:type / arr:type_array
type = QM v:$("string" / "number" / "integer" / "object" / "array" / "boolean" / "null") QM

kw_enum = QM key:"enum" QM name_separator value:array
kw_const = QM key:"const" QM name_separator value:value
kw_default = QM key:"default" QM name_separator value:value


```

```
// ---------- Keywords annotation ----------
annotation_keyword = (kws_annotation_stringValues / kw_examples / kws_annotation_booleanValues)

kws_annotation_stringValues = QM key:$("title"/"description"/"$comment") QM name_separator
    value:string
kw_examples = QM key:"examples" QM name_separator value:array
kws_annotation_booleanValues = QM key:$("readOnly"/"writeOnly"/"deprecated") QM name_separator
    value:boolean

// ---------- Keywords string ----------
string_keyword = kws_string_length / kw_pattern / kw_format

kws_string_length = QM key:$("minLength" / "maxLength") QM name_separator value:int
kw_pattern = QM key:"pattern" QM name_separator value:pattern_string

kw_format = QM key:"format" QM name_separator value:format_value
format_value = QM f:("date-time" / "time" / "date" / "duration" / "email" / "idn-email" /
    "hostname" / "idn-hostname" / "ipv4" / "ipv6" / "uuid" / "uri-reference" / "uri-template" /
    "uri" / "iri-reference" / "iri" / "json-pointer" / "relative-json-pointer" / "regex") QM

// ---------- Keywords number ----------
number_keyword = kw_multipleOf / kws_range

kw_multipleOf = QM key:"multipleOf" QM name_separator value:positiveNumber
kws_range = QM key:$("minimum" / "exclusiveMinimum" / "maximum" / "exclusiveMaximum") QM
    name_separator value:number

// ---------- Keywords object ----------
object_keyword = kws_props / kw_moreProps / kw_requiredProps / kw_propertyNames / kws_size

kws_props = QM key:$("patternProperties"/"properties") QM name_separator value:object_schemaMap
kw_moreProps = QM key:$("additionalProperties"/"unevaluatedProperties") QM name_separator
    value:schema_object
kw_requiredProps = QM key:"required" QM name_separator value:string_array
kw_propertyNames = QM key:"propertyNames" QM name_separator value:schema_object
kws_size = QM key:$("minProperties" / "maxProperties") QM name_separator value:int

// ---------- Keywords array ----------
array_keyword = kw_items / kw_prefixItems / kw_unevaluatedItems / kw_contains / kws_mContains /
    kws_array_length / kw_uniqueness

kw_items = QM key:"items" QM name_separator value:schema_object
kw_prefixItems = QM key:"prefixItems" QM name_separator value:schema_array
kw_unevaluatedItems = QM key:"unevaluatedItems" QM name_separator value:schema_object
kw_contains = QM key:"contains" QM name_separator value:schema_object
kws_mContains = QM key:$("minContains" / "maxContains") QM name_separator value:int
kws_array_length = QM key:$("minItems" / "maxItems") QM name_separator value:int
```

```
kw_uniqueness = QM key:"uniqueItems" QM name_separator value:boolean

// ---------- Keywords media ----------
media_keyword = kw_contentMediaType / kw_contentSchema / kw_contentEncoding

kw_contentMediaType = QM key:"contentMediaType" QM name_separator value:string
kw_contentSchema = QM key:"contentSchema" QM name_separator value:schema_object

kw_contentEncoding = QM key:"contentEncoding" QM name_separator value:encoding
encoding = QM e:$("7bit"/"8bit"/"binary"/"quoted-printable"/"base16"/"base32"/"base64") QM

// ---------- Keywords schema composition ----------
schemaComposition_keyword = kws_combineSchemas / kw_notSchema

kws_combineSchemas = QM key:$("allOf"/"anyOf"/"oneOf") QM name_separator value:schema_array
kw_notSchema = QM key:"not" QM name_separator value:schema_object

// ---------- Keywords conditional subschemas ----------
conditionalSubschemas_keyword = kw_dependentRequired / kw_dependentSchemas / kw_ifThenElse

kw_dependentRequired = QM key:"dependentRequired" QM name_separator
    value:object_arrayOfStringsMap
kw_dependentSchemas = QM key:"dependentSchemas" QM name_separator value:object_schemaMap
kw_ifThenElse = QM key:$("if" / "then" / "else") QM name_separator value:schema_object

// ---------- Keywords structuring ----------
structuring_keyword = kw_schema / kw_id / kw_anchor / kw_ref / kw_defs

kw_schema = QM key:"$schema" QM name_separator value:schema_value
schema_value = QM v:"https://json-schema.org/draft/2020-12/schema" QM

kw_id = QM key:"$id" QM name_separator value:schema_id
kw_anchor = QM key:"$anchor" QM name_separator value:anchor
kw_ref = QM key:"$ref" QM name_separator value:schema_ref
kw_defs = QM key:"$defs" QM name_separator value:object_schemaMap

// ----- Objects -----
schema_object = boolean /
                ws "{" ws members:(head:keyword tail:(value_separator m:keyword)*)? ws "}" ws

object = begin_object members:(head:member tail:(value_separator m:member)*)? end_object
member = name:string name_separator value:value

object_schemaMap = begin_object members:(head:schema_member tail:(value_separator
    m:schema_member)*)? end_object
schema_member = name:string name_separator value:schema_object

```

```
object_arrayOfStringsMap = begin_object members:(head:arrayOfStrings_member
    tail:(value_separator m:arrayOfStrings_member)*)? end_object
arrayOfStrings_member = name:string name_separator value:string_array

// ----- Arrays -----
array = begin_array values:(head:value tail:(value_separator v:value)*)? end_array

string_array = begin_array values:(head:string tail:(value_separator v:string)*)? end_array

schema_array = begin_array values:(head:schema_object tail:(value_separator v:schema_object)*)
    end_array

type_array = begin_array values:(head:type tail:(value_separator v:type)*)? end_array

// ----- Numbers -----
number "number" = "-"? int frac?
positiveNumber "positive number" = ("0" frac / [1-9] [0-9]* frac?)

exp = [eE] ("-"/"+")? [0-9]+
frac = "." [0-9]+

int "integer" = integer:(("0"* i:([1-9] [0-9]*)) / (i:"0" "0"*))

// ----- Strings -----
string "string" = QM str:$char* QM
pattern_string = QM str:$[^"]* QM
anchor "anchor" = QM value:anchor_value QM
schema_id = QM "https:/" "/datagen.di.uminho.pt"? id:$("/schemas" ("/" [^/#"]+)+) QM
schema_ref "$ref" = QM "https://datagen.di.uminho.pt"?
    ref:$(("/schemas/" [^/#"]+)? ref_segment / "/schemas/" [^/#"]+) QM

anchor_value = $([a-zA-Z][a-zA-Z0-9\-\_\:\.]*)
ref_segment = "#" (anchor_value / ("/" [^/#"]+)*)

char = unescaped
  / escape sequence:( '"' / "\\" / "/" / "b" / "f" / "n" / "r" / "t" / "u" digits:$(HEXDIG
    HEXDIG HEXDIG HEXDIG))

escape = "\\"
QM = '"'

unescaped = [^\0-\x1F\x22\x5C]
HEXDIG = [0-9a-f]i

// ---------- Keyword datagen ----------
datagen_keyword = QM key:"_datagen" QM name_separator QM f:(func:"pattern" args:pattern_arg /
    func:datagen_func args:datagen_args?) QM
```

```
datagen_func = datagen_boolean / datagen_integer / datagen_float / datagen_string

datagen_boolean = func:"boolean"
datagen_integer = func:("index" / "integerOfSize" / "integer")
datagen_float = func:("float" / "multipleOf")
datagen_string = func:("date" / "formattedInteger" / "formattedFloat" / "guid" / "hexBinary" /
    "language" / "letter" / "lorem" / "objectID" / "position" / "pt_phone_number" /
    "stringOfSize" / "time" / "xsd_dateTime" / "xsd_date" / "xsd_duration" / "xsd_gDay" /
    "xsd_gMonthDay" / "xsd_gMonth" / "xsd_gYearMonth" / "xsd_gYear" / "xsd_string" / "actor" /
    "animal" / "brand" / "buzzword" / "capital" / "car_brand" / "continent" / "country" /
    "cultural_center" / "firstName" / "fullName" / "gov_entity" / "hacker" / "job" / "month" /
    "musician" / "nationality" / "political_party_abbr" / "political_party_name" /
    "pt_businessman" / "pt_city" / "pt_county" / "pt_district" / "pt_entity_abbr" /
    "pt_entity_name" / "pt_parish" / "pt_politician" / "pt_public_figure" /
    "pt_top100_celebrity" / "religion" / "soccer_club" / "soccer_player" / "sport" / "surname" /
    "top100_celebrity" / "weekday" / "writer")

datagen_args = "(" datagen_args_content? datagen_args_close ws
datagen_args_content = (!datagen_args_close). datagen_args_content*
datagen_args_close = ")"

pattern_arg = "('" pattern_arg_content? pattern_arg_close ws
pattern_arg_content = (!pattern_arg_close). pattern_arg_content*
pattern_arg_close = "')"
```

C

JSON SCHEMA INVERTER

```
function notSubschema(json) {
    let notTypes = [];

    for (let t in json.type) {
        if (!Object.keys(json.type[t]).length) { notTypes.push(t); delete json.type[t]; }
        else {
            notGenericKeys(json.type[t]);
            switch (t) {
                case "number": notNumeric(json.type[t]); break;
                case "string": notString(json.type[t]); break;
                case "object": notObject(json.type[t]); break;
                case "array": notArray(json.type[t]); break;
            }
        }
    }

    if (!Object.keys(json.type).length) {
        let types = ["string","number","object","array","null","boolean"];
        types = types.filter(t => !notTypes.includes(t));

        if (types.length > 1 && types.includes("null")) types.splice(types.indexOf("null"), 1);
        for (let t of types) json.type[t] = {};
    }
}

function notGenericKeys(json) {
    if ("const" in json) {
        json.notValues = json.const;
        delete json.const;
    }
    if ("enum" in json) {
        if ("notValues" in json) json.notValues = json.notValues.concat(json.enum);
        else json.notValues = json.enum;
        delete json.enum;
    }
```

```
    if ("default" in json) {
        json.notDefault = json.default;
        delete json.default;
    }
}

function notNumeric(json) {
    let invertSchema = (old_k, new_k) => {
        let value = json[old_k];
        Object.keys(json).map(k => delete json[k]);
        json[new_k] = value;
    }

    if ("integer" in json) {
        if (json.integer) json.integer = false;
        else delete json.integer;
    }
    if ("mininum" in json) invertSchema("minimum", "exclusiveMaximum");
    else if ("exclusiveMinimum" in json) invertSchema("exclusiveMinimum", "maximum");
    else if ("maximum" in json) invertSchema("maximum", "exclusiveMinimum");
    else if ("exclusiveMaximum" in json) invertSchema("exclusiveMaximum", "minimum");
    else {
        let {multipleOf, notMultipleOf} = json;

        if (multipleOf !== undefined && notMultipleOf !== undefined) {
            let temp = multipleOf;
            json.multipleOf = notMultipleOf;
            json.notMultipleOf = temp;
        }
        else if (multipleOf !== undefined) {
            json.notMultipleOf = multipleOf;
            delete json.multipleOf;
        }
        else if (notMultipleOf !== undefined) {
            json.multipleOf = notMultipleOf;
            delete json.notMultipleOf;
        }
    }
    return json;
}

function notString(json) {
    if ("pattern" in json) json.pattern = `^((?!(${json.pattern})).){${"minLength" in json ?
    json.minLength : 10},${"maxLength" in json ? json.maxLength : 30}}`;
    if ("format" in json) json.notFormat = [json.format];
    notSizeKeys(json, "minLength", "maxLength");
}
```

```
function notObject(json) {
    notProperties(json, ["properties","patternProperties"]);
    notOtherElements(json, ["additionalProperties","unevaluatedProperties"]);
    if ("required" in json && !("properties" in json)) {
        json.notRequired = json.required;
        delete json.required;
    }
    if ("propertyNames" in json) notString(json.propertyNames.type.string);
    notSizeKeys(json, "minProperties", "maxProperties");
}

function notArray(json) {
    notOtherElements(json, ["items","unevaluatedItems"]);
    notSizeKeys(json, "minItems", "maxItems");
    if ("uniqueItems" in json) json.uniqueItems = !json.uniqueItems;

    if ("prefixItems" in json) {
        for (let i = 0; i < json.prefixItems.length; i++) notSubschema(json.prefixItems[i]);
    }

    if ("contains" in json) {
        let notContainsTypes = [];

        for (let i = 0; i < json.contains.length; i++) {
            if (json.contains[i].maxContains !== null) {
                json.contains[i].minContains = json.contains[i].maxContains + 1;
                json.contains[i].maxContains = null;
            }
            else if (json.contains[i].minContains !== null) {
                json.contains[i].maxContains = json.contains[i].minContains - (!json.contains[i
    ].minContains ? 0 : 1);
                json.contains[i].minContains = null;
            }
            else {
                let c = json.contains.splice(i--, 1)[0].contains;

                for (let t in c.type) {
                    if (!Object.keys(c.type[t]).length) {
                        if (!notContainsTypes.includes(t)) notContainsTypes.push(t);
                        delete c.type[t];
                    }
                }

                if (Object.keys(c.type).length > 0) {
                    notSubschema(c);
                    if ("notContains" in json) json.notContains.push(c);
                    else json.notContains = [c];
```

```
                }
            }
        }

        if (!json.contains.length) delete json.contains;
        if (notContainsTypes.length > 0) {
            if ("notContainsTypes" in json) json.notContainsTypes = json.notContainsTypes.concat
    (notContainsTypes.filter(t => !json.notContainsTypes.includes(t)));
            else json.notContainsTypes = notContainsTypes;
        }
    }
}

function notSizeKeys(json, min, max) {
    if (max in json) {
        json[min] = json[max] + 1;
        delete json[max];
    }
    else if (min in json) {
        json[max] = json[min] - (!json[min] ? 0 : 1);
        delete json[min];
    }
}

function notProperties(json, keys) {
    keys.filter(k => k in json).map(k => {
        for (let p in json[k]) {
            notSubschema(json[k][p]);
            if (!Object.keys(json[k][p]).length) delete json[k][p];
        }
    })
}

function notOtherElements(json, keys) {
    keys.filter(k => k in json).map(k => {
        if (json[k] === false) json[k] = {type: {string: {}, number: {}, boolean: {}, null: {},
    array: {}, object: {}}};
        else {
            let notTypes = Object.keys(json[k].type);
            notSubschema(json[k]);

            notTypes = notTypes.filter(x => !Object.keys(json[k].type).includes(x));
            if (!Object.keys(json[k].type).length) json[k] = false;
            else json["not" + k.charAt(0).toUpperCase() + k.slice(1)] = notTypes;
        }
    })
}
```

```
module.exports = {
    notGenericKeys,
    notNumeric,
    notString,
    notObject,
    notArray
}
```

D

JSON SCHEMA EXTENDER

```
const inverter = require('./schema_inverter');

function extendSchema(json, schema, type, key, SETTINGS) {
    if (key == "not") {
        inverter.notGenericKeys(schema);
        switch (type) {
            case "number": inverter.notNumeric(schema); break;
            case "string": inverter.notString(schema); break;
            case "object": inverter.notObject(schema); break;
            case "array": inverter.notArray(schema); break;
        }
    }

    extendArrayKey(json, schema, ["const","enum","default","notValues","notDefault"]);
    switch (type) {
        case "number": extendNumeric(json, schema); break;
        case "string": extendString(json, schema); break;
        case "object": extendObject(json, schema, SETTINGS); break;
        case "array": extendArray(json, schema, SETTINGS); break;
    }
}

function extendArrayKey(json, schema, keys) {
    keys.filter(k => k in schema).map(key => {
        if (key in json)
            json[key] = json[key].concat(schema[key].filter(x => !json[key].includes(x)));
        else json[key] = schema[key];
    })
}

function extendString(json, schema) {
    if ("pattern" in schema) json.pattern = schema.pattern;
    if ("format" in schema) json.format = schema.format;
    extendArrayKey(json, schema, ["notFormat"]);
    extendSizeKeys(json, schema, "minLength", "maxLength");
```

```
}

function extendNumeric(json, schema) {
    let {minimum, maximum, exclusiveMinimum, exclusiveMaximum} = schema;
    if ("integer" in schema) json.integer = schema.integer;

    extendArrayKey(json, schema, ["multipleOf","notMultipleOf"]);

    if (minimum !== undefined) {
        if ("minimum" in json) {
            if (minimum > json.minimum) json.minimum = minimum;
        }
        else if ("exclusiveMinimum" in json) {
            if (minimum > json.exclusiveMinimum) {
                json.minimum = minimum;
                delete json.exclusiveMinimum;
            }
        }
        else json.minimum = minimum;

        if ("maximum" in json && json.minimum > json.maximum) delete json.maximum;
        else if ("exclusiveMaximum" in json && json.minimum >= json.exclusiveMaximum)
            delete json.exclusiveMaximum;
    }
    else if (exclusiveMinimum !== undefined) {
        if ("minimum" in json) {
            if (exclusiveMinimum >= json.minimum) {
                json.exclusiveMinimum = exclusiveMinimum;
                delete json.minimum;
            }
        }
        else if ("exclusiveMinimum" in json) {
            if (exclusiveMinimum > json.exclusiveMinimum)
                json.exclusiveMinimum = exclusiveMinimum;
        }
        else json.exclusiveMinimum = exclusiveMinimum;

        if ("maximum" in json && json.exclusiveMinimum >= json.maximum) delete json.maximum;
        else if ("exclusiveMaximum" in json && json.exclusiveMinimum >= json.exclusiveMaximum)
            delete json.exclusiveMaximum;
    }

    if (maximum !== undefined) {
        if ("maximum" in json) {
            if (maximum < json.maximum) json.maximum = maximum;
        }
        else if ("exclusiveMaximum" in json) {
```

```
            if (maximum < json.exclusiveMaximum) {
                json.maximum = maximum;
                delete json.exclusiveMaximum;
            }
        }
        else json.maximum = maximum;

        if ("minimum" in json && json.maximum < json.minimum) delete json.minimum;
        else if ("exclusiveMinimum" in json && json.maximum <= json.exclusiveMinimum)
            delete json.exclusiveMinimum;
    }
    else if (exclusiveMaximum !== undefined) {
        if ("maximum" in json) {
            if (exclusiveMaximum <= json.maximum) {
                json.exclusiveMaximum = exclusiveMaximum;
                delete json.maximum;
            }
        }
        else if ("exclusiveMaximum" in json) {
            if (exclusiveMaximum < json.exclusiveMaximum)
                json.exclusiveMaximum = exclusiveMaximum;
        }
        else json.exclusiveMaximum = exclusiveMaximum;

        if ("minimum" in json && json.exclusiveMaximum <= json.minimum) delete json.minimum;
        else if ("exclusiveMinimum" in json && json.exclusiveMaximum <= json.exclusiveMinimum)
            delete json.exclusiveMinimum;
    }
}

function extendObject(json, schema, SETTINGS) {
    assignProperties(json, schema, ["properties","patternProperties"],
        SETTINGS.extend_objectProperties);
    assignSchemaObject(json, schema, ["additionalProperties","unevaluatedProperties",
        "propertyNames"], SETTINGS.extend_schemaProperties);
    extendSizeKeys(json, schema, "minProperties", "maxProperties");
    extendArrayKey(json, schema, ["required","notRequired","notAdditionalProperties",
        "notUnevaluatedProperties"]);
}

function extendArray(json, schema, SETTINGS) {
    assignSchemaObject(json, schema, ["items","unevaluatedItems"],
        SETTINGS.extend_schemaProperties);
    extendSizeKeys(json, schema, "minItems", "maxItems");
    extendArrayKey(json, schema, ["contains","notContains","notContainsTypes","notItems",
        "notUnevaluatedItems"]);
    if ("uniqueItems" in schema) json.uniqueItems = schema.uniqueItems;
```

```
    if ("prefixItems" in schema) {
        let setting = SETTINGS.extend_prefixItems;

        if ("prefixItems" in json && setting != "OWT") {
            if (/^O/.test(setting)) {
                for (let i = 0; i < schema.prefixItems.length; i++) {
                    if (i < json.prefixItems.length) {
                        // OR - extend schemas at the same index
                        if (setting == "OR") assignSubschema(json.prefixItems[i],
    schema.prefixItems[i]);
                        // OWP - overwrite only schemas at the same index
                        if (setting == "OWP") json.prefixItems[i] = schema.prefixItems[i];
                    }
                    else json.prefixItems.push(schema.prefixItems[i]);
                }
            }
            // AP - append new prefixItems to old one
            if (setting == "AP") json.prefixItems = json.prefixItems.concat(schema.prefixItems);
        }
        // OWT - overwrite old prefixItems completely with the new one
        else json.prefixItems = schema.prefixItems;
    }
}

function extendSizeKeys(json, schema, min, max) {
    if (min in schema) {
        if (min in json) {
            if (schema[min] > json[min]) json[min] = schema[min];
        }
        else json[min] = schema[min];

        if (max in json && json[max] < json[min]) delete json[max];
    }

    if (max in schema) {
        if (max in json) {
            if (schema[max] < json[max]) json[max] = schema[max];
        }
        else json[max] = schema[max];

        if (min in json && json[min] > json[max]) delete json[min];
    }
}

function assignProperties(json, schema, keys, setting) {
    keys.filter(k => k in schema).map(k => {
```

```
        if (k in json) {
            for (let p in schema[k]) {
                if (p in json[k] && setting == "OR") assignSubschema(json[k][p], schema[k][p]);
                else json[k][p] = schema[k][p];
            }
        }
        else json[k] = schema[k];
    })
}

function assignSchemaObject(json, schema, keys, setting) {
    keys.filter(k => k in schema).map(k => {
        if (k in json) {
            if (typeof json[k] == "boolean" || typeof schema[k] == "boolean")
                json[k] = schema[k];
            else {
                if (setting == "OR") assignSubschema(json[k], schema[k]);
                else json[k] = schema[k];
            }
        }
        else json[k] = schema[k];
    })
}

function assignSubschema(json, schema) {
    for (let t in schema.type) {
        if (t in json.type && Object.keys(schema.type[t]).length > 0)
            extendSchema(json.type[t], schema.type[t], t, null);
        else json.type[t] = schema.type[t];
    }
}

module.exports = { extendSchema }
```

E

XML SCHEMA GRAMMAR (CODE OMITTED)

```
DSL_text = ws dec:XML_declaration xsd:schema comments

ws "whitespace" = [ \t\n\r]*
ws2 = [ \t\n\r]+

// ----- XML declaration -----
XML_declaration = comments dec:$("<?xml" XML_version XML_encoding? XML_standalone? ws '?>') ws

XML_version = ws2 "version" ws "=" ws q1:QM "1.0" q2:QM

XML_encoding = ws2 "encoding" ws "=" ws q1:QM XML_encoding_value q2:QM
XML_encoding_value = "UTF-"("8"/"16") / "ISO-10646-UCS-"("2"/"4") / "ISO-8859-"[1-9] / "ISO
    -2022-JP" / "Shift_JIS" / "EUC-JP"

XML_standalone = ws2 "standalone" ws "=" ws q1:QM XML_standalone_value q2:QM
XML_standalone_value = "yes" / "no"

// ----- <schema> -----
schema = comments open_XSD_el el_name:"schema" attrs:schema_attrs ws ">" ws
    content:schema_content close_schema

close_schema = prefix:close_XSD_prefix "schema" ws ">" ws

schema_attrs = attrs:(formDefault / blockDefault / finalDefault / xmlns / elem_id / elem_lang /
    schema_version / targetNamespace)+

formDefault = ws2 attr:$(("attribute"/"element")"FormDefault") ws "=" q1:QMo val:form_values
    q2:QMc
blockDefault = ws2 attr:"blockDefault" ws "=" q1:QMo val:block_values q2:QMc
finalDefault = ws2 attr:"finalDefault" ws "=" q1:QMo val:finalDefault_values q2:QMc
xmlns = ws2 "xmlns" prefix:(":" p:NCName {return p})? ws "=" ws val:string
schema_version = ws2 attr:"version" ws "=" ws val:string
targetNamespace = ws2 attr:"targetNamespace" ws "=" ws val:string
```

```
schema_content = el:((redefine / include / import / annotation)* (((simpleType / complexType /
    group / attributeGroup) / element / attribute / notation) annotation*)*)

// ----- <include> -----
include = comments prefix:open_XSD_el el_name:"include" attrs:schemaLocID_attrs ws
    close:(merged_close / ann_content)

schemaLocID_attrs = el:(schemaLocation elem_id? / elem_id schemaLocation?)?

schemaLocation = ws2 attr:"schemaLocation" ws "=" ws val:string

// ----- <import> -----
import = comments prefix:open_XSD_el el_name:"import" attrs:import_attrs ws
    close:(merged_close / ann_content)

import_attrs = el:(import_namespace / elem_id / schemaLocation)*

import_namespace = ws2 attr:"namespace" ws "=" ws val:string

// ----- <redefine> -----
redefine = comments prefix:open_XSD_el el_name:"redefine" attrs:schemaLocID_attrs ws
    close:(merged_close / openEl content:redefine_content close_el:close_XSD_el)

redefine_content = c:(comments annotation/ (simpleType / complexType / group / attributeGroup))*

// ----- <element> -----
element = comments prefix:open_XSD_el el_name:"element" attrs:element_attrs ws
    close:(merged_close / openEl content:element_content close_el:close_XSD_el)

element_attrs = el:(elem_abstract / elem_block / elem_default / elem_substitutionGroup /
                elem_final / elem_fixed / elem_form / elem_id / elem_minOccurs /
                elem_maxOccurs / elem_name / elem_nillable / elem_ref / elem_type)*

elem_abstract = ws2 attr:"abstract" ws "=" q1:QMo val:boolean q2:QMc
elem_block = ws2 attr:"block" ws "=" q1:QMo val:block_values q2:QMc
elem_default = ws2 attr:"default" ws "=" ws val:string
elem_final = ws2 attr:"final" ws "=" q1:QMo val:elem_final_values q2:QMc
elem_fixed = ws2 attr:"fixed" ws "=" ws val:string
elem_form = ws2 attr:"form" ws "=" q1:QMo val:form_values q2:QMc
elem_id = ws2 attr:"id" ws "=" q1:QMo val:ID q2:QMc
elem_maxOccurs = ws2 attr:"maxOccurs" ws "=" q1:QMo val:(int/"unbounded") q2:QMc
elem_minOccurs = ws2 attr:"minOccurs" ws "=" q1:QMo val:int q2:QMc
elem_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
elem_nillable = ws2 attr:"nillable" ws "=" q1:QMo val:boolean q2:QMc
elem_lang = ws2 attr:"xml:lang" ws "=" q1:QMo val:language q2:QMc
elem_ref = ws2 attr:"ref" ws "=" q1:QMo val:QName q2:QMc
elem_source = ws2 attr:"source" ws "=" ws val:string
```

```
elem_substitutionGroup = ws2 attr:"substitutionGroup" ws "=" q1:QMo val:QName q2:QMc
elem_type = ws2 attr:"type" ws "=" q1:QMo val:type_value q2:QMc

element_content = c:(datagen_comment? comments annotation? (simpleType / complexType)?
    (keyOrUnique / keyref)*)

// ----- <field> -----
field = comments prefix:open_XSD_el el_name:"field" attrs:field_attrs ws
    close:(merged_close / ann_content)

field_attrs = attrs:(field_xpath elem_id? / elem_id field_xpath?)?

field_xpath = ws2 attr:"xpath" ws "=" q1:QMo val:fieldXPath q2:QMc

// ----- <selector> -----
selector = comments prefix:open_XSD_el el_name:"selector" attrs:selector_attrs ws
    close:(merged_close / ann_content)

selector_attrs = attrs:(selector_xpath elem_id? / elem_id selector_xpath?)?

selector_xpath = ws2 attr:"xpath" ws "=" q1:QMo val:selectorXPath q2:QMc

// ----- <key/unique> -----
keyOrUnique = comments prefix:open_XSD_el el_name:$("key"/"unique") attrs:keyOrUnique_attrs ws
    close:(merged_close / openEl content:xpath_content close_el:close_XSD_el)

keyOrUnique_attrs = attrs:(elem_constraint_name elem_id? / elem_id elem_constraint_name?)?

elem_constraint_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc

xpath_content = c:(comments annotation? (selector field+))

// ----- <keyref> -----
keyref = comments prefix:open_XSD_el el_name:"keyref" attrs:keyref_attrs ws
    close:(merged_close / openEl content:xpath_content close_el:close_XSD_el)

keyref_attrs = attrs:(elem_id / elem_constraint_name / keyref_refer)*

keyref_refer = ws2 attr:"refer" ws "=" q1:QMo val:QName q2:QMc

// ----- <attribute> -----
attribute = comments prefix:open_XSD_el el_name:"attribute" attrs:attribute_attrs ws
    close:(merged_close / openEl content:attribute_content close_el:close_XSD_el)

attribute_attrs = el:(elem_default / elem_fixed / elem_form / elem_id / attr_name / attr_ref /
    elem_type / attr_use)*

```

```
attr_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
attr_ref = ws2 attr:"ref" ws "=" q1:QMo val:QName q2:QMc
attr_use = ws2 attr:"use" ws "=" q1:QMo val:use_values q2:QMc

attribute_content = c:(datagen_comment? comments annotation? simpleType?)

// ----- <attributeGroup> -----
attributeGroup = comments prefix:open_XSD_el el_name:"attributeGroup" attrs:attributeGroup_attrs
     ws close:(merged_close / openEl content:attributeGroup_content close_el:close_XSD_el)

attributeGroup_attrs = el:(elem_id / attrGroup_name / attrGroup_ref)*

attrGroup_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
attrGroup_ref = ws2 attr:"ref" ws "=" q1:QMo val:QName q2:QMc

attributeGroup_content = c:(comments annotation? attributes)

// ----- <anyAttribute> -----
anyAttribute = comments prefix:open_XSD_el el_name:"anyAttribute" attrs:anyAttribute_attrs ws
    close:(merged_close / ann_content)

anyAttribute_attrs = el:(elem_id / any_namespace / processContents)*

any_namespace = ws2 attr:"namespace" ws "=" ws val:namespace_values
processContents = ws2 attr:"processContents" ws "=" q1:QMo val:processContents_values q2:QMc

// ----- <any> -----
any = comments prefix:open_XSD_el el_name:"any" attrs:any_attrs ws
    close:(merged_close / ann_content)

any_attrs = el:(elem_id / elem_maxOccurs / elem_minOccurs / any_namespace / processContents)*

// ----- <simpleType> -----
simpleType = comments prefix:open_XSD_el el_name:"simpleType" attrs:simpleType_attrs ws openEl
    ws content:simpleType_content close_el:close_XSD_el

simpleType_attrs = el:(simpleType_final / elem_id / simpleType_name)*

simpleType_final = ws2 attr:"final" ws "=" q1:QMo val:simpleType_final_values q2:QMc
simpleType_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc

simpleType_content = c:(comments annotation? (restrictionST / list / union))

// ----- <annotation> -----
annotation = comments prefix:open_XSD_el el_name:"annotation" attr:elem_id? ws
    close:(merged_close / openEl content:annotation_content close_el:close_XSD_el)

```

```
annotation_content = (appinfo / documentation)*

// ----- <appinfo> -----
appinfo = comments (appinfo_simple / appinfo_prefix) comments

appinfo_simple = "<" el_name:"appinfo" attr:elem_source? ws close:("/>" ws / openEl
    content:appinfo_content_simple? close_appinfo_simple)

appinfo_prefix = prefix:open_XSD_el el_name:"appinfo" attr:elem_source? ws close:(merged_close /
     openEl content:appinfo_content_prefix? close_el:close_appinfo_prefix)

appinfo_content_simple = comments (!close_appinfo_simple). appinfo_content_simple*
appinfo_content_prefix = comments (!close_appinfo_prefix). appinfo_content_prefix*

close_appinfo_simple = "</appinfo" ws ">" ws
close_appinfo_prefix = prefix:close_XSD_prefix name:"appinfo" ws ">" ws

// ----- <documentation> -----
documentation = comments (doc_simple / doc_prefix) comments

documentation_attrs = attrs:(elem_source elem_lang? / elem_lang elem_source?)?

doc_simple = "<" el_name:"documentation" attrs:documentation_attrs ws close:("/>" ws /
    openEl content:doc_content_simple? close_doc_simple)
doc_prefix = prefix:open_XSD_el el_name:"documentation" attrs:documentation_attrs ws
    close:(merged_close / openEl content:doc_content_prefix? close_el:close_doc_prefix)

doc_content_simple = comments (!close_doc_simple). doc_content_simple*
doc_content_prefix = comments (!close_doc_prefix). doc_content_prefix*

close_doc_simple = "</documentation" ws ">" ws
close_doc_prefix = prefix:close_XSD_prefix name:"documentation" ws ">" ws

// ----- <union> -----
union = comments prefix:open_XSD_el el_name:"union" attrs:union_attrs ws close:(merged_close /
    openEl content:union_content close_el:close_XSD_el)

union_attrs = attrs:(elem_id union_memberTypes? / union_memberTypes elem_id?)?

union_memberTypes = ws2 attr:"memberTypes" ws "=" q1:QMo val:list_types q2:QMc

union_content = c:(comments annotation? simpleType*)

// ----- <list> -----
list = comments prefix:open_XSD_el el_name:"list" attrs:list_attrs ws close:(merged_close /
    openEl content:list_content close_el:close_XSD_el)

```

```
list_attrs = attrs:(elem_id list_itemType? / list_itemType elem_id?)?

list_itemType = ws2 attr:"itemType" ws "=" q1:QMo val:type_value q2:QMc

list_content = c:(comments annotation? simpleType?)

// ----- <restriction> (simpleType) -----
restrictionST = comments prefix:open_XSD_el el_name:"restriction" attrs:base_attrs ws
    close:(merged_close / openEl content:restrictionST_content close_el:close_XSD_el)

base_attrs = attrs:(base elem_id? / elem_id base?)?

base = ws2 attr:"base" ws "=" q1:QMo val:type_value q2:QMc

restrictionST_content = comments h1:annotation? h2:simpleType? t:constrFacet*

// ----- <restriction> (simpleContent) -----
restrictionSC = comments prefix:open_XSD_el el_name:"restriction" attrs:base_attrs ws
    close:(merged_close / openEl content:restrictionSC_content close_el:close_XSD_el)

restrictionSC_content = comments c:(restrictionST_content attributes)

// ----- <restriction> (complexContent) -----
restrictionCC = comments prefix:open_XSD_el el_name:"restriction" attrs:base_attrs ws
    close:(merged_close / openEl content:CC_son_content close_el:close_XSD_el)

CC_son_content = c:(comments annotation? (all / choiceOrSequence / group)? attributes)

// ----- <extension> (simpleContent) -----
extensionSC = comments prefix:open_XSD_el el_name:"extension" attrs:base_attrs ws
    close:(merged_close / openEl content:extensionSC_content close_el:close_XSD_el)

extensionSC_content = c:(comments annotation? attributes)

// ----- <extension> (complexContent) -----
extensionCC = comments prefix:open_XSD_el el_name:"extension" attrs:base_attrs ws
    close:(merged_close / openEl content:CC_son_content close_el:close_XSD_el)

// ----- <minExclusive/minInclusive/maxExclusive/maxInclusive/totalDigits/fractionDigits/length/
    minLength/maxLength/enumeration/whiteSpace/pattern> -----
constrFacet = comments prefix:open_XSD_el el_name:constrFacet_values attrs:constrFacet_attrs ws
    close:(merged_close / ann_content)

constrFacet_attrs = el:(elem_id / constrFacet_fixed / constrFacet_value)*

constrFacet_fixed = ws2 attr:"fixed" ws "=" q1:QMo val:boolean q2:QMc
constrFacet_value = ws2 attr:"value" ws "=" ws val:string
```

```
257
// ----- <complexType> -----
259 complexType = comments prefix:open_XSD_el el_name:"complexType" attrs:complexType_attrs ws
    close:(merged_close / openEl content:complexType_content close_el:close_XSD_el)
261
complexType_attrs = el:(elem_abstract / complexType_block / elem_final / elem_id / complex_mixed
     / complexType_name)*
263
complexType_block = ws2 attr:"block" ws "=" q1:QMo val:elem_final_values q2:QMc
265 complex_mixed = ws2 attr:"mixed" ws "=" q1:QMo val:boolean q2:QMc
complexType_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
267
complexType_content = c:(comments annotation? (simpleContent / complexContent / ((all /
    choiceOrSequence / group)? attributes)))
269
// ----- <simpleContent> -----
271 simpleContent = comments prefix:open_XSD_el el_name:"simpleContent" attr:elem_id? ws openEl
    content:simpleContent_content close_el:close_XSD_el

273 simpleContent_content = c:(comments annotation? (restrictionSC / extensionSC))

275 // ----- <complexContent> -----
complexContent = comments prefix:open_XSD_el el_name:"complexContent" attrs:complexContent_attrs
     ws openEl content:complexContent_content close_el:close_XSD_el
277
complexContent_attrs = attrs:(complex_mixed elem_id? / elem_id complex_mixed?)?
279
complexContent_content = c:(comments annotation? (restrictionCC / extensionCC))
281
// ----- <all> -----
283 all = comments prefix:open_XSD_el el_name:"all" attrs:all_attrs ws close:(merged_close /
    openEl content:all_content close_el:close_XSD_el)
285
all_attrs = el:(elem_id / all_maxOccurs / all_minOccurs)*
287
all_maxOccurs = ws2 attr:"maxOccurs" ws "=" q1:QMo val:"1"  q2:QMc
289 all_minOccurs = ws2 attr:"minOccurs" ws "=" q1:QMo val:[01] q2:QMc

291 all_content = c:(comments annotation? element*)

293 // ----- <choice/sequence> -----
choiceOrSequence = comments prefix:open_XSD_el el_name:$("choice"/"sequence")
295     attrs:choiceOrSeq_attrs ws close:(merged_close / openEl content:choiceOrSeq_content
    close_el:close_XSD_el)
297
choiceOrSeq_attrs = el:(elem_id / elem_maxOccurs / elem_minOccurs)*
299 choiceOrSeq_content = c:(comments annotation? (element / choiceOrSequence / group / any)*)
```

```
// ----- <group> -----
group = comments prefix:open_XSD_el el_name:"group" attrs:group_attrs ws close:(merged_close / openEl content:group_content close_el:close_XSD_el)

group_attrs = el:(group_name / elem_id / elem_maxOccurs / elem_minOccurs / group_ref)*

group_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
group_ref = ws2 attr:"ref" ws "=" q1:QMo val:QName q2:QMc

group_content = c:(comments annotation? (all / choiceOrSequence)?)

// ----- <notation> -----
notation = comments prefix:open_XSD_el el_name:"notation" attrs:notation_attrs ws
    close:(merged_close / ann_content)

notation_attrs = el:(elem_id / notation_name / notation_URI_attrs)*

notation_name = ws2 attr:"name" ws "=" q1:QMo val:NCName q2:QMc
notation_URI_attrs = ws2 attr:("public" / "system") ws "=" ws val:string

// ----- Comment -----
comments = comment*
comment = "<!--" comment_content close_comment ws
comment_content = (!close_comment). comment_content*
close_comment = "-->"

// ----- DataGen type -----
datagen_comment = "<!--datagen:" func:datagen_func args:datagen_args? "-->" ws
datagen_func = datagen_boolean / datagen_integer / datagen_float / datagen_string

datagen_boolean = func:"boolean"
datagen_integer = func:("index"/"integer"/"integerOfSize")
datagen_float = func:("float"/"multipleOf")
datagen_string = func:("date"/"formattedInteger"/"formattedFloat"/"guid"/"hexBinary"/"language"/
    "letter"/"lorem"/"objectID"/"pattern"/"position"/"pt_phone_number"/"stringOfSize"/"time"/
    "xsd_date"/"xsd_dateTime"/"xsd_duration"/"xsd_gDay"/"xsd_gMonth"/"xsd_gMonthDay"/"xsd_gYear"
    /"xsd_gYearMonth"/"xsd_string"/"actor"/"animal"/"brand"/"buzzword"/"capital"/"car_brand"/
    "continent"/"country"/"cultural_center"/"firstName"/"fullName"/"gov_entity"/"hacker"/"job"/
    "month"/"musician"/"nationality"/"political_party_abbr"/"political_party_name"/
    "pt_businessman"/"pt_city"/"pt_county"/"pt_district"/"pt_entity_abbr"/"pt_entity_name"/
    "pt_parish"/"pt_politician"/"pt_public_figure"/"pt_top100_celebrity"/"religion"/
    "soccer_club"/"soccer_player"/"sport"/"surname"/"top100_celebrity"/"weekday"/"writer")

datagen_args = "(" datagen_args_content? datagen_args_close ws
datagen_args_content = (!datagen_args_close). datagen_args_content*
datagen_args_close = ")"
```

```
// ----- Recurring regex -----
openEl  = ">" ws
closeEl = ">" ws

open_XSD_el      = "<"  prefix:(p:NCName ":")?
close_XSD_prefix = "</" prefix:(p:NCName ":")?

merged_close = "/>" ws comments

close_XSD_el = prefix:close_XSD_prefix name:XSD_el_name ws closeEl comments
ann_content = openEl comments content:annotation? close_el:close_XSD_el

attributes = c:((attribute / attributeGroup)* anyAttribute?)

// ----- Values -----
QM = '"' / "'"
QMo = ws qm:('"' / "'") ws
QMc = ws qm:('"' / "'")

boolean = true / false
false = "false"
true  = "true"
null  = "null"

int = integer:(("0"* i:([1-9] [0-9]*)) / (i:"0" "0"*))

letter = [a-zA-Z]
letter1_8 = $(letter letter? letter? letter? letter? letter? letter? letter?)
string = ('"'[^"]*'"' / "'"[^']*"'")

NCName = $(([a-zA-Z_]/[^\x00-\x7F])([a-zA-Z0-9.\-_]/[^\x00-\x7F])*)
QName = prefix:(p:NCName ":")? name:NCName

ID = id:NCName
language = $((letter letter / [iI]"-"letter+ / [xX]"-"letter1_8)("-"letter1_8)?)

XSD_el_name = "include"/"import"/"redefine"/"notation"/"annotation"/"appinfo"/"documentation"/
    "element"/"field"/"selector"/"key"/"keyref"/"unique"/"attributeGroup"/"attribute"/
    "anyAttribute"/"simpleType"/"union"/"list"/"restriction"/"extension"/constrFacet_values/
    "complexType"/"simpleContent"/"complexContent"/"all"/"choice"/"group"/"sequence"/"any"

// ----- Simple values of attributes -----
form_values = $("un"?"qualified")
use_values = "optional" / "prohibited" / "required"
processContents_values = "lax" / "skip" / "strict"
constrFacet_values = $("length" / ("max"/"min")"Length" / ("max"/"min")("Ex"/"In")"clusive" /
    ("total"/"fraction")"Digits" / "whiteSpace" / "pattern" / "enumeration")
```

```
type_value = type:(p:NCName ":" name:NCName / name:NCName)

// ----- Lists of values of attributes -----
finalDefault_values = "#all" / finalDefault_listOfValues
finalDefault_list_val = "extension" / "restriction" / "list" / "union"
finalDefault_listOfValues = l:$(finalDefault_list_val (ws2 finalDefault_list_val)*)

elem_final_values = "#all" / "extension" ws "restriction" / "restriction" ws "extension" /
    "extension" / "restriction"

list_types = ws fst:type_value? others:(ws2 n:type_value)* ws

block_values = "#all" / block_listOfValues
block_list_val = "extension" / "restriction" / "substitution"
block_listOfValues = l:$(block_list_val (ws2 block_list_val)*)

simpleType_final_values = "#all" / simpleType_final_listOfValues
simpleType_final_list_val = "list" / "union" / "restriction"
simpleType_final_listOfValues = l:$(simpleType_final_list_val (ws2 simpleType_final_list_val)*)


namespace_values = (namespace_values_Q / namespace_values_A)
namespace_values_Q = $('"' ws ("##any" / "##other" / l:namespace_listOfValues_Q) ws '"')
namespace_values_A = $("'" ws ("##any" / "##other" / l:namespace_listOfValues_A) ws "'")

namespace_list_val_Q = "##local" / "##targetNamespace" / $((!("##"/'"')). [^ "\t\n\r]+)
namespace_list_val_A = "##local" / "##targetNamespace" / $((!("##"/"'")). [^ '\t\n\r]+)

namespace_listOfValues_Q = $(namespace_list_val_Q (ws2 namespace_list_val_Q)*)
namespace_listOfValues_A = $(namespace_list_val_A (ws2 namespace_list_val_A)*)

// ----- XPath -----
selectorXPath = $(path ('|' path)*)
path = ('.//')? step ('/' step)*
fieldXPath = $(('.//')? (step '/')* (step / '@' nameTest))
step = '.' / nameTest
nameTest = QName / '*' / NCName ':' '*'
```