Synthetic Data Generation from JSON Schemas

Hugo André Coelho Cardoso
University of Minho, Braga, Portugal

José Carlos Ramalho

Department of Informatics, University of Minho, Braga, Portugal

Abstract

This document describes the steps taken in the development of DataGen From Schemas. This new version of DataGen is an application that makes it possible to automatically generate representative synthetic datasets from JSON and XML schemas, in order to facilitate tasks such as the thorough testing of software applications and scientific endeavors in relevant areas, namely Data Science. This paper focuses solely on the JSON Schema component of the application.

DataGen's prior version is 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 specification to a DataGen model, then using the original application as a middleware to generate the final datasets.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Theory of computation → Grammars and context-free languages; Information systems → Open source software

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

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.5

1 Introduction

The current landscape of the techonological and 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 [5] present themselves as difficult obstacles to the growth of these sciences and to the progression of many projects in the development [4] and testing phases.

In this context, the generation of synthetic data arises as a possible solution for these problems, 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

© Hugo André Coelho Cardoso and José Carlos Ramalho;
licensed under Creative Commons License CC-BY 4.0
11th Symposium on Languages, Applications and Technologies (SLATE 2022).
Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 5; pp. 5:1–5:16

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1993 [6], 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 [10]), 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 [2], spacial microsimulation models [8], Internet of Things (IoT) [1], Deep Learning [3] and automotive applications [9].

This paper intends to expose and document the processes of ideation and development of the application DataGen From Schemas, specifically its JSON Schema component, whose objective is to generate synthetic datasets directly from JSON 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 the following section.

2 DataGen

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 APIs 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.

It is strongly encouraged to check the published paper on DataGen's development and functionality [7] first, 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.

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 its 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 datasets to test and debug platforms in development, as well as 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 JSON schemas. By doing this, the user is given the option to specify the structure of the intended dataset in JSON 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 JSON 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. The intended workflow for this JSON Schema component is depicted in the following image:

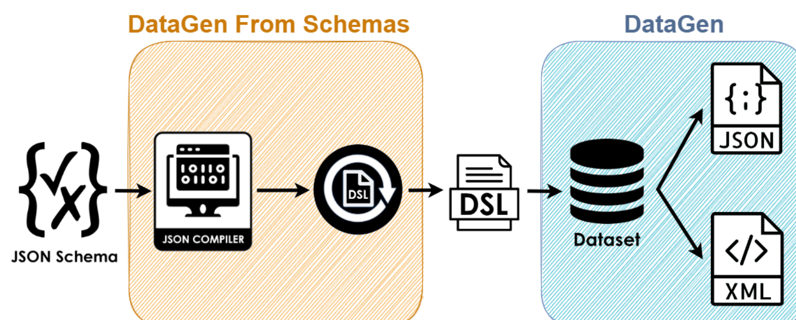

Figure 1 Intended workflow for the JSON Schema component.

The program will accept the user's input in the form of a JSON schema, which will then be parsed by a PEG.js grammar-based 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.

DataGen's prior version was created by the same authors of this new tool, so there is access to the original application. The goal is to place the new compilers and translators in DataGen's own backend, in order to centralize the functionalities, avoiding additional requests between servers and possible downtime, as would be the case if DataGen From Schemas was deployed in an entirely separate web application and had to communicate with DataGen's API routes via HTTP requests to generate datasets from its models.

3 JSON Format

This section aims to give a bit of contextualization about the format JSON, in order to have a better grasp of what DataGen From Schemas must be able to analyse and parse, what data is revelant to be extracted, and also to understand the limits and potential of this simple, yet versatile format and why it was chosen for this application, instead of another of many popular data formats nowadays, i.e. its relevance and adequacy to this particular project. As mentioned in section 2, DataGen already has the ability of generating JSON and XML instances from DSL models, so this new product's focus is uniquely translating the data in a JSON schema to a DSL model, basically the opposite procedure.

What will be exposed about JSON also applies to JSON Schema, as the latter is a JSON vocabulary, which is to say, 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.

JavaScript Object Notation (JSON) is a lightweight data exchange format derived from the programming language Javascript, whose simplicity and readability contributed to its current vast popularity. It has ample and growing support in countless programming languages (C++, Java, JavaScript, Python, etc), which make it a strong candidate for fast and compact exchange of information between applications.

This format is utilized to represent structured information, i.e. data with rigid and well-defined configurations, where no divergencies are allowed from the structures established in the schema, namely a different data type for a certain field. This reflects in the JSON syntax, where there are only four different primitive types for values - string, number, boolean or null - and the instances are built upon solely two different data structures: objects (non-ordered collections of key-value pairs) and arrays (ordered lists of values).

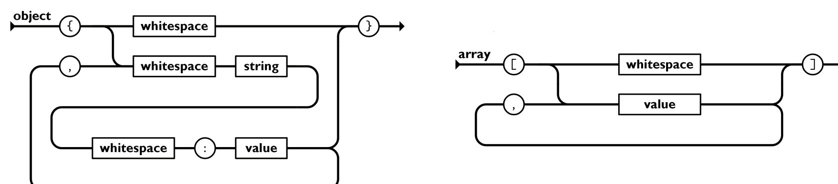

Figure 2 Structures of a JSON object and array.

DataGen's own DSL is built upon the JSON format, so there is very high compatibility with JSON Schema and a guarantee that all kinds of data specified in the schema can be represented in the model. In addition, DataGen also has a tool named interpolation functions, which makes it possible to randomize the value of a certain field, given its type and some other restrictions. With this, it is possible to convert a JSON schema into a corresponding non-deterministic DSL model, where the final values are not hard-coded, but decided in runtime, which may result in endless different valid datasets from a single schema.

4 Development

In order to generate DSL models from JSON schemas, DataGen From Schemas will need two different components: firstly, a grammar-based parser to analyze the schema and extract all useful information. For this, PEG.js was the technology of choice, as it was already used in DataGen and proved adequate and capable of parsing JSON-like structures. Lastly, a program

capable of translating the intermediate structure into a DSL model, for which JavaScript was used, for direct compatibility with PEG.js (that also incorporates this programming language) and JSON.

The user may input one or more schemas into the program, which is necessary to allow cross-schema referencing. In this case, the user must indicate which is the primary schema from which the dataset is to be generated. The parser then analyzes all schemas sequentially, building an intermediate structure for each of them, a procedure that will be explained in detail in this section.

4.1 Grammar

The grammar was built with a JSON grammar available in the PEG.js Github as its foundation, given that JSON Schema uses the same syntax. 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 document, JSON Schema 2020-12.

As such, the aforementioned JSON grammar was modified into a dialect (a specific version of JSON Schema) grammar 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. With this, it was possible to implement other 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*. 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. This validation only does not work with references, since these are only resolved posteriorly;
- **error reports** – following the previous points, whenever the parser finds an error, be it a syntatic or semantic error, it halts the execution of the pipeline and reports it to the users, for them to correct and try again.

The other central point of the grammar is building the intermediate data structure, to where it extracts the relevant information. Before addressing this topic, it is best to categorize and explain the different kinds of JSON Schema keywords. It 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 used:

- **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 schema 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 non-JSON data). The parser recognizes these keywords but willingly ignores them, since they have little to no use in a dataset generation context.

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 to them and only them (the aforementioned type-oriented keywords), so it is easy to separate most keywords by type;
- a single JSON schema may validate against multiple data types - the same schema can have keywords respective to arrays, numbers, and strings, for example. As such, it is useful to know, at all times, exactly what data types are produceable from the schema;
- 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;
- 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. Then, it separates the keyword's values by type and introduces one 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:

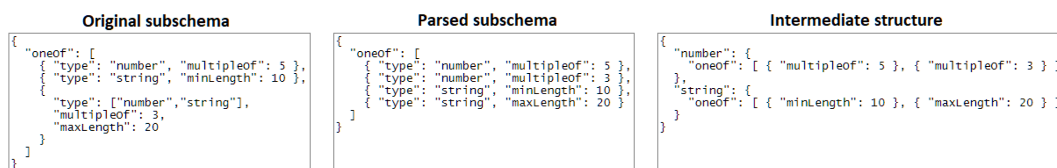

Figure 3 Example of parsing done on a schema composition keyword.

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 data structure to store all relevant data, in a way designed to facilitate and make more efficient the following translator program's 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 understand their schema better and detect unwilling errors.

4.2 Referencing

JSON Schema references can vary a lot: there are absolute and relative references, depending on if they include the schema's base URI. It is not mandatory for a schema to have an *id*, which is its URI-reference, but without one, it is unable to be referenced by 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 the previous subsection, 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 as it is reading the schemas, since they might be pointing to schemas or subschemas that have yet to be reached. For the sake of consistency and 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 it is ultimately 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.

4.3 Creating the DSL model

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 an according DSL string, taking into account how they influence each other. For the sake of brevity, the keywords will not be explained minutely in this paper, 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 joining 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. 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:

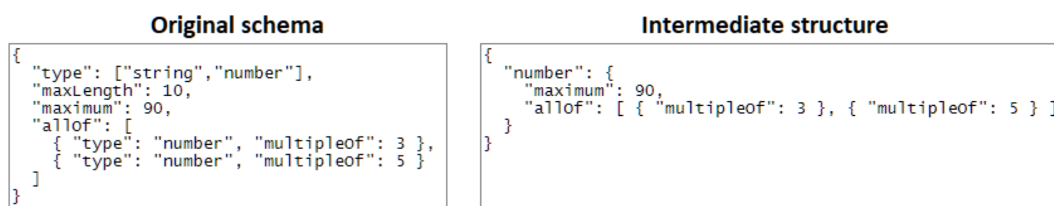

Figure 4 Example of parsing done on a apparent multi-type schema.

The program then randomly selects one of the generateable types and moves on to translating its keywords. The first step is to resolve any keywords of schema composition or conditional application of subschemas that there may be - 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 understandable 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 would result in values 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 the restraints specified in these keywords. 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 these subschemas in the extension process, so that ultimately the base schema is extended only with type-specific keywords.

4.3.1 Schema composition keywords

There are four keywords belonging to this category: *allOf*, *anyOf*, and *oneOf* 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* declares that an instance must not be valid against the given subschema.

For the first three, a subset of their values’ schemas 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 (as was described in subsection 4.3), 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 to manually and sequentially extend a base schema with each type-specific keyword of others. 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.

Due to its complexity, the functionality of these two components will not be addressed in detail in this document. The reader can find their explanation in another more substantial and exhaustive academic paper on the entirety of DataGen From Schemas that will be published in the near future.

4.3.2 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:

```
{
  "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 5 Example of a schema with dynamic semantics.

In this case, the schema will only know what schema 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.

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 4.4.3.

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.

4.4 Translating the intermediate structure

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. In this subsection, it will be carefully detailed the method behind translating each type of value.

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 unallowed 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 imutable. 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 goes on to actually translate the type's-oriented 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", "amber", "green", null, 42] }
{ "const": "United States of America" }
{ "type": "null" }
{ "type": "boolean" }
```

```
gen => { return gen.random(...["red","amber","green",null,42]) }
gen => { return gen.random(...["United States of America"]) }
null
'{{boolean()}}'
```

Figure 6 Translation of the keywords *enum* and *const*.

4.4.1 String type

The string type only has four different keywords: *pattern*, *format*, and *minLength*/*maxLength*. Only in this case, it was decided that there would be an order of precedence to these keywords since, realistically speaking, they would 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 one of DataGen's interpolation functions, *format* is more contrived, since there is a need to program a different DSL string for each format, to generate an according value, some of which map directly to existing interpolation functions and others that do not. Below are presented some examples of the models generated from this data type's keywords:

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

{
  "type": "string",
  "minLength": 2,
  "maxLength": 3
}

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

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


'{{stringOfSize(2, 3)}}'


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

Figure 7 Translation of string type schemas.

4.4.2 Numeric types

As for numeric types, there are five applicable 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 a multiple of 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 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), in order to obtain values that belong to the intended range. This is doable by rounding down the result of the division of the upper bound by the LCM and rounding up the result of the division of the lower bound by the LCM, respectively. If the schema only has either an upper or lower range boundary, the other one is offset by 100 units to provide confortable margin for generateable values. At this point, there are three different possible outcomes, detailed next and further illustrated in the image below:

- 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 bounds and multiplying it by the LCM.

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

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

{
  "type": "integer",
  "multipleOf": 2.5,
  "minimum": 0,
  "exclusiveMaximum": 20
}
```

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

'{{multipleOf(7)}}'


gen => { return gen.integer(0,3) * 5 }
```

Figure 8 Translation of numeric type schemas.

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

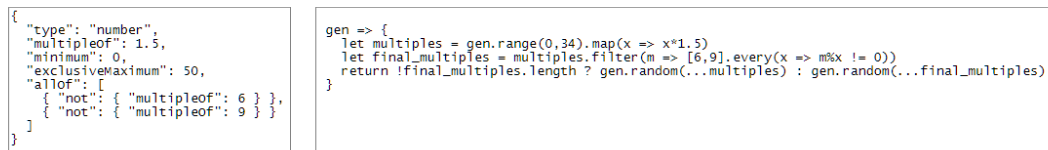

Figure 9 Translation of a complex numeric type schema.

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 its 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.

4.4.3 Object type

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

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

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

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 value's schema, storing the pair in the intermediate object structure. 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 sequentially, producing the DSL strings of their respective values and storing the pairs in the intermediate object;
- iterates the value of the keyword *patternProperties* - for each pair, there is a probability (customizable by the user) to produce, at most, one according instance property, 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* and *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 specified, *additionalProperties*'s schema prevails, else it is the only used keyword's. The program translates this schema into a DSL string and generates random names for the properties keys, either according to the *propertyNames* schema, if present, or by generating small chunks of *lorem ipsum*.

Having finalized the intermediate object with each property's DSL string, these properties are all joined in a string encased by curly braces, in the syntax of a regular JavaScript object that DataGen is able to parse and then generate the according dataset.

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" },
    "type": { "enum": ["residential", "business"] }
  },
  "required": ["street_address", "city", "state"],
  "propertyNames": { "pattern": "inhabitant[1-3]" },
  "additionalProperties": { "enum": ["John","James","Mary"] },
  "minProperties": 5,
  "maxProperties": 7
}
```

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

Figure 10 Translation of an object type schema.

4.4.4 Array type

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

For this data type, the program creates an intermediate array structure firstly, in order to maleably plot the intended array, and only at the end converts it to a DSL string. 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 intermediate array as is.

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 intermediate array, in order.

Then, DataGen From Schemas parses the inclusion keywords. In the workflow of this solution, these keywords efectively 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 with 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 values validated against the inclusion keys to be some of the prefixed items, then the program cannot guarantee this.

Once all the "hard-coded" elements are in the intermediate array, 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*, their schemas are inverted (using the schema inverter mentioned in 4.3.1) and used to extend the schema for additional items, in order to guarantee that all additional items wholly conform to the user's configuration. This final schema is used to sequentially generate a random amount of additional items' DSL strings, within the intended range for the array, which are appended to the intermediate structure.

Finally, DataGen From Schemas checks if all these items must be unique or not (*uniqueItems*). This keyword, as is the case with the inclusion keys, demands a compromise and an inteligent workaround, since it is impossible to check for the items' uniqueness before generating them, and is the reason that warrants the usage of an intermeadiate array to store the DSL strings of each element, instead of simply building a single string and appending each item to it. As such, depending on the value of this keyword, two different types of DSL strings for the final array may be created:

- **not unique items** – if either the keyword is not specified or its value is false, then the procedure is to simply create a string of an array, where its elements are the DSL strings stored in the intermediate 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 11 Translation of an array type schema with non-unique items.

- **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 different. As can be observed 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 value of the latest try is pushed to the array anyway, to prevent crashing the program, and the resulting array ends up not obbeying 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}
      }
    }
  return arr
}
```

Figure 12 Translation of an array type schema with unique items.

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. 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 is still possible to satisfy some use cases.

5 Conclusion

The main contributions of this paper are the design of a synthetic data generator from JSON schemas and its implementation, which culminate in DataGen From Schemas - a quick and sturdy solution built upon DataGen that effectively greatly expands the base apllication's functionality and makes it more accessible to any user, by allowing their input to be in a vastly popular and utilized vocabulary, JSON Schema, and automating the creation of DataGen's DSL model.

With this software, it becomes possible to very quickly specify a dataset and generate several different instances of it, given that the program randomizes its values with each attempt, as well as further customizing its requirements directly in DataGen, via the provided intermediate DSL model, with support from custom datasets and other tools provided by the base application, which is not possible to do in JSON Schema's syntax.

DataGen From Schemas has been experimented with real-life cases and convoluted schemas that use mechanisms such as bundling or recursion, and proves to be an adequate solution, capable of interpreting the schemas it is given and generating representative datasets accordingly. The product will soon be put in a production environment and made available for the general public, as a free and open-source application, after an arduous yet successful development phase.

References

1 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.

2 Jessamyn Dahmen and Diane Cook. Synsys: A synthetic data generation system for healthcare applications. *Sensors*, 19(5), 2019. doi:10.3390/s19051181.

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

4 GAO. Artificial intelligence in health care: Benefits and challenges of machine learning in drug development (staa)-policy briefs & reports-epta network. In *GAO Technology Assessment: Artificial Intelligence in Health Care: Benefits and Challenges of Machine Learning in Drug Development*, 2020. Accessed: 2021-04-25. URL: https://eptanetwork.org/database/policy-briefs-reports/1898-artificial-intelligence-in-health-care-benefits-and-challenges-of-machine-learning-in-drug-development-staa.

5 Menno Mostert, Annelien Bredenoord, Monique Biesaart, and Johannes Delden. Big data in medical research and eu data protection law: challenges to the consent or anonymise approach. *European Journal of Human Genetics*, 24:1096–1096, July 2016. doi:10.1038/ejhg.2016.71.

6 Donald B. Rubin. Statistical disclosure limitation. In *Journal of Official Statistics*, pages 461–468, 1993.

7 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)*, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.SLATE.2021.6.

8 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.

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

10 P. Voigt and A. von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer International Publishing, 2017. URL: https://books.google.pt/books?id=cWAwDwAAQBAJ.