



Sorbonne Université

LIP6 - Laboratoire d'Informatique Sorbonne

Rapport de Stage : Rétro-ingénierie du générateur DataGen

Stagiaire : Abdelkader Boumessaoud

Équipe :

Abdelkader Boumessaoud, Sorbonne Université, abdelkader.boumessaoud@etu.sorbonne-universite.fr

Attouche Lyes, Université Paris-Dauphine, lyes.attouche@dauphine.fr

Mohamed-Salim Aissi, Sorbonne Université, saissi1701@gmail.com

Tuteur de stage / Coordinateur :

Mohamed-Amine Baazizi, LIP6, mohamed-amine.baazizi@lip6.fr

Établissement : Sorbonne Université

Organisme d'accueil : LIP6 - Laboratoire d'Informatique de Sorbonne Université

15 juin 2023 - 04 août 2023

Table des matières

1	Contexte et Objectif	2
2	JSON Schema	2
2.1	Description	2
2.1.1	Exemple d'un schéma décrivant un objet	2
2.2	Validation de données JSON	3
2.2.1	Exemple de validation d'une instance	3
3	Génération d'instances optimistes pour JSON-Schema	5
3.1	Description du problème	5
3.2	Approches	6
4	Étude et compréhension du générateur	6
4.1	Workflow d'un composant JSON Schema	6
4.1.1	Étape 0 : <i>Schema d'entrée</i>	6
4.1.2	Étape 1 : <i>Schema étendu</i>	6
4.1.3	Étape 2 : <i>DSL</i>	7
4.1.4	Étape 3 : <i>Dataset</i>	7
4.1.5	Étape 4 : <i>Instance finale</i>	7
4.2	Fonctionnement des types spécifiques	7
4.2.1	string	7
4.2.2	number, integer	8
4.3	Fonctionnement des objects	8
4.4	Fonctionnement des arrays	9
4.5	Fonctionnement des connecteurs logiques	10
4.5.1	Conjonction (<i>allOf</i>)	10
4.5.2	Disjonction (<i>oneOf</i>)	11
4.5.3	Disjonction (<i>anyOf</i>)	11
4.5.4	Négation (<i>not</i>)	12
4.6	Explication du fonctionnement de l'extension de schéma	12
4.7	Explication du fonctionnement de la négation de schéma	13
4.8	Explication du modèle DSL	14
4.9	Explication du modèle Dataset pré XML/JSON	14
5	Étude des limitations de DataGen	14
5.1	Choix de la librairie	14
5.1.1	Prise en main et installation du générateur DataGen	15
5.2	Choix du validateur JSON Schema	15
5.3	Approche adoptée	16
5.4	Expérimentations	16
5.4.1	Étapes suivies	16
5.4.2	Collections de schémas utilisés	17
5.4.3	Classement des résultats	18
5.5	Remarques	18
6	Contraintes	18
6.1	Linguistique	18
6.2	Flux de traitement intégré	19
6.3	Matériel	19
6.4	Logiciel	19
6.5	Débogage	19
7	Perspectives	19
8	Conclusion	20

1 Contexte et Objectif

JSON Schema est un standard bien connu pour décrire la structure des données JSON, et son utilisation se généralise dans divers domaines tels que la description d'API, la vérification de pipelines d'apprentissage et les métadonnées, entre autres.

Dans le cadre de ce stage, nous nous intéressons spécifiquement au générateur **DataGen** [6], dont l'objectif est de générer automatiquement des instances de tests à partir de schémas JSON donnés en entrée. Cette approche est cruciale pour gérer divers cas d'utilisation, tels que l'interopérabilité des microservices et la validation de données JSON dans les systèmes de bases de données.

Dans ce stage, notre principal objectif est de caractériser les limites de **DataGen**. Nous adoptons une approche de rétro-ingénierie manuelle combinée à une analyse expérimentale pour évaluer **DataGen**. Nous souhaitons identifier les classes de schémas JSON que **DataGen** traite correctement, ainsi que celles qui posent des problèmes.

Cette analyse nous permettra également d'identifier les opportunités d'amélioration de **DataGen** et de définir les limites de son approche optimiste. En résumé, ce stage vise à apporter un éclairage critique sur les performances du générateur automatique d'instances de tests **DataGen**, tout en explorant des pistes d'amélioration pour mieux répondre aux besoins de l'industrie et des applications.

2 JSON Schema

2.1 Description

Le langage est basé sur le format JSON et utilise des mots clés dédiés pour définir des contraintes sur la structure des données. Ces mots-clés sont regroupés en familles et permettent de décrire pour chaque type de données les informations suivantes :

- Pour les nombres, il est possible de contraindre leur intervalle de valeur en utilisant les mots-clés **minimum** et **maximum**. On peut également imposer qu'ils soient multiples d'un certain nombre avec le mot-clé **multipleOf**.
- Pour les chaînes de caractères, on dispose de plusieurs possibilités. Le mot-clé **minLength** permet de spécifier la longueur minimale d'une chaîne, tandis que **maxLength** définit la longueur maximale. On peut également imposer des motifs de caractères à respecter en utilisant le mot-clé **pattern**. De plus, il est possible de définir une liste de valeurs acceptées avec le mot-clé **enum**.
- Pour les objets, on peut décrire les propriétés attendues en utilisant **required** et préciser le type de chaque propriété en utilisant **properties** et **patternProperties**. On peut imposer une longueur au moyen de **minProperties** et **maxProperties**.
- Pour les tableaux, le mot-clé **items** est utilisé pour définir le type des éléments du tableau. On peut également spécifier la taille minimale et maximale du tableau avec les mots-clés **minItems** et **maxItems**, ainsi que d'autres contraintes telles que l'existence d'un élément satisfaisant une contrainte **contains**.
- L'assertion **type** permet d'imposer le type de l'instance à valider. Par défaut, les autres assertions expriment une implication implicite.

2.1.1 Exemple d'un schéma décrivant un objet

Le schéma suivant décrit une instance de type objet qui, pour être valide, doit contenir trois propriétés obligatoires : **firstName**, **lastName** et **age**. De plus, les propriétés **firstName** et **lastName** doivent être des chaînes de caractères de longueur supérieure ou égale à 3.

```
1 {  
2   "type": "object",  
3   "properties": {  
4     "firstName": {  
5       "type": "string",  
6       "minLength": 3  
7     },  
8     "lastName": {  
9       "type": "string",  
10      "minLength": 3
```

```
11     },
12     "age": {
13         "type": "integer",
14         "minimum": 18
15     }
16 },
17 "required": ["age", "firstName", "lastName"]
18 }
```

2.2 Validation de données JSON

La validation d'une instance sur un schéma est relativement simple. Elle a été spécifiée par le comité de standardisation et consiste à retourner les erreurs de validations de manière exhaustive. Ces erreurs peuvent concerner les objets (propriétés manquantes ou non conformes), les types de base (valeurs en dehors des plages autorisées), les tableaux (absence de valeurs ou valeurs non conformes). Le standard définit une sortie avec plusieurs niveaux de détails : un booléen, un niveau détaillés aplatis et détaillé hiérarchique.

2.2.1 Exemple de validation d'une instance

Schéma + Instance non valide car le champ `age` est censé être de type `number`, mais sa valeur est une chaîne de caractères (`thirty`). Le champ `zipcode` est censé être de type `number`, mais sa valeur est une chaîne de caractères (`"ABCDE"`).

```
1 {
2     "type": "object",
3     "properties": {
4         "name": {
5             "type": "string"
6         },
7         "age": {
8             "type": "number"
9         },
10        "address": {
11            "type": "object",
12            "properties": {
13                "street": {
14                    "type": "string"
15                },
16                "city": {
17                    "type": "string"
18                },
19                "zipcode": {
20                    "type": "number"
21                }
22            },
23            "required": [
24                "street",
25                "city",
26                "zipcode"
27            ]
28        },
29    },
30    "required": [
31        "name",
32        "age",
33        "address"
34    ]
35 }
```

```

1 {
2   "name": "John Doe",
3   "age": "thirty",
4   "address": {
5     "street": "123 Main St.",
6     "city": "New York",
7     "zipcode": "ABCDE"
8   }
9 }

```

Résultats de la validation (Output format : *Hierarchical*)

```

1 {
2   "valid": false,
3   "evaluationPath": "",
4   "schemaLocation": "https://json-everything.net/e351761edb",
5   "instanceLocation": "",
6   "details": [
7     {
8       "valid": true,
9       "evaluationPath": "/properties/name",
10      "schemaLocation": "https://json-everything.net/e351761edb#/properties/name",
11      "instanceLocation": "/name"
12    },
13    {
14      "valid": false,
15      "evaluationPath": "/properties/age",
16      "schemaLocation": "https://json-everything.net/e351761edb#/properties/age",
17      "instanceLocation": "/age",
18      "errors": {
19        "type": "Value is \"string\" but should be \"number\""
20      }
21    },
22    {
23      "valid": false,
24      "evaluationPath": "/properties/address",
25      "schemaLocation": "https://json-everything.net/e351761edb#/properties/address",
26      "instanceLocation": "/address",
27      "details": [
28        {
29          "valid": true,
30          "evaluationPath": "/properties/address/properties/street",
31          "schemaLocation":
↪ "https://json-everything.net/e351761edb#/properties/address/properties/street",
32          "instanceLocation": "/address/street"
33        },
34        {
35          "valid": true,
36          "evaluationPath": "/properties/address/properties/city",
37          "schemaLocation":
↪ "https://json-everything.net/e351761edb#/properties/address/properties/city",
38          "instanceLocation": "/address/city"
39        },
40        {
41          "valid": false,
42          "evaluationPath": "/properties/address/properties/zipcode",
43          "schemaLocation":
↪ "https://json-everything.net/e351761edb#/properties/address/properties/zipcode",
44          "instanceLocation": "/address/zipcode",
45          "errors": {
46            "type": "Value is \"string\" but should be \"number\""

```

```

47     }
48   }
49 ]
50 }
51 ]
52 }

```

Dans cet exemple, la validation a commencé à la racine de l'instance JSON, et trois éléments de la propriété **details** indiquent les erreurs de validation spécifiques. Le premier élément indique que la propriété **name** de l'instance JSON est valide, tandis que le deuxième élément indique que la propriété **age** est invalide. La propriété **errors** de cet élément indique que la valeur de **age** est une chaîne de caractères alors qu'elle devrait être un nombre.

Le troisième élément de **details** indique que la propriété **address** est invalide, mais elle contient également des éléments **details** pour ses propriétés **street**, **city**, et **zipcode**. Les deux premières propriétés sont valides, tandis que la troisième propriété **zipcode** est invalide avec la même erreur que **age**.

3 Génération d'instances optimistes pour JSON-Schema

3.1 Description du problème

La génération d'instances consiste à obtenir, à partir d'un schéma, un ensemble d'instances valides pour ce schéma. Garantir une génération correcte est complexe à cause de l'expressivité du langage. Or, dans de nombreux cas les schémas sont simples, avec un processus de génération tout aussi simple. C'est cette hypothèse qu'utilisent les générateurs optimistes. Nous verrons que ces approches ne tiennent pas compte de toutes les contraintes pour éviter une explosion combinatoire lors de la génération.

Un exemple de l'expressivité du langage est le schéma ci-dessous :

```

1 {
2   "type": "object",
3   "properties": {
4     "prop": {
5       "allOf": [
6         {
7           "enum": [
8             "forbidden",
9             "mandatory"
10          ]
11        },
12        {
13          "not": {
14            "enum": [
15              "forbidden"
16            ]
17          }
18        }
19      ]
20    }
21  },
22  "required": [
23    "prop"
24  ]
25 }

```

Ici, on a deux contraintes (donc deux propositions logiques) qui doivent être conjuguées par le générateur, ainsi qu'une négation, qui doit être correctement interprétée. Cet aspect logique de JSON Schema rend la génération d'instances parfois compliquée : l'outil doit être capable d'interpréter des

contraintes comme propositions logiques, et d'effectuer dessus des conjonctions, disjonctions, ou négations. Cela s'avère un problème difficile du point de vue de sa programmation, étant donné les possibilités pléthoriques du langage en termes de contraintes (propriétés, expressions régulières, types...)

3.2 Approches

Nous nous sommes concentrés sur trois bibliothèques, toutes publiées en tant que logiciel libre, déjà identifiées et choisies pour leur compatibilité avec la quasi-totalité des opérateurs de JSON Schema :

json-schema-faker Une bibliothèque JavaScript [3] de génération de données fictives pour les documents JSON, qui utilise des générateurs de données aléatoires. Elle se base sur la spécification JSON Schema pour définir le contenu autorisé d'un document JSON, et permet de générer des données basiques ou complexes conformes au schéma donné en entrée.

DataGen Une application qui permet de générer automatiquement des ensembles de données synthétiques à partir de schémas JSON et XML. L'objectif principal de **DataGen** est de faciliter des tâches telles que le test de logiciels et les projets scientifiques dans des domaines pertinents tels que la science des données.

4 Étude et compréhension du générateur

4.1 Workflow d'un composant JSON Schema

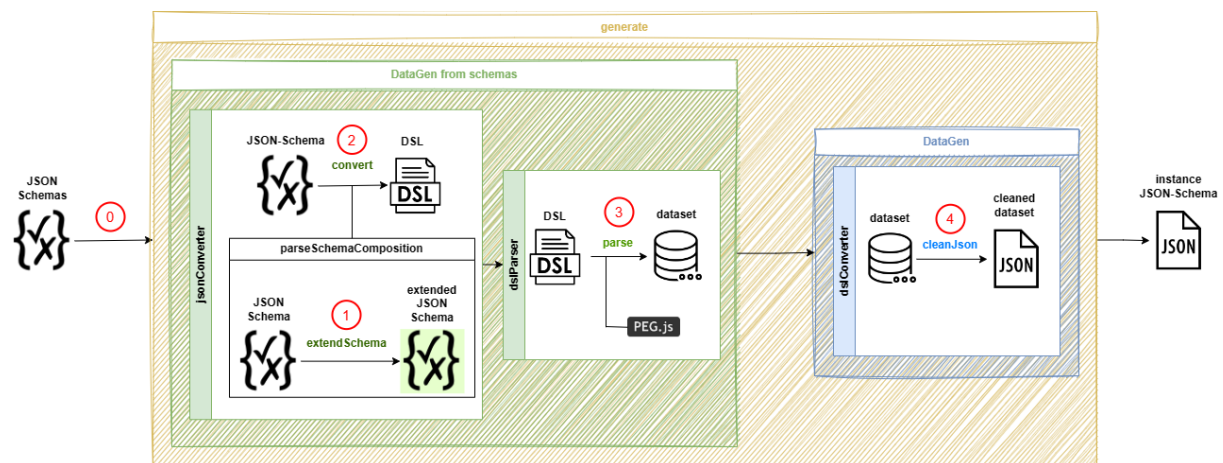


Figure 1 : Workflow pour composant JSON Schema

4.1.1 Étape 0 : *Schema d'entrée*

L'utilisateur peut entrer un ou plusieurs schémas dans le programme, ce qui est nécessaire pour permettre référencement croisé de schémas. Dans ce cas, l'utilisateur doit indiquer quel est le schéma principal à partir duquel le jeu de données doit être généré.

4.1.2 Étape 1 : *Schema étendu*

L'analyseur analyse ensuite tous les schémas de manière séquentielle pour construire une structure intermédiaire pour chacun d'eux, cette structure est une structure orientée par type, où chaque valeur du schéma est décrite par un objet JavaScript qui mappe chacun de ses types de données créables à leurs mots-clés respectifs, et les valeurs sont organisées dans une structure hiérarchique.

schema_extender (Extension de schéma) : L'extension d'un schéma se fait en analysant les mots-clés de composition de schéma ou d'application conditionnelle de sous-schémas, tels que "allOf", "anyOf", "oneOf" et "not", permettent de combiner plusieurs schémas en un seul schéma plus complexe.

Avant de générer des données à partir du schéma, **DataGen** analyse ces mots-clés et étend le schéma de base avec leur contenu. Par exemple, si le schéma contient le mot-clé "allOf", cela signifie que l'instance doit valider tous les schémas spécifiés dans le tableau correspondant. Dans ce cas, **DataGen** étend le schéma de base avec les contraintes de tous les schémas spécifiés dans le tableau.

De même, si le schéma contient le mot-clé "anyOf", cela signifie que l'instance doit valider au moins un des schémas spécifiés dans le tableau correspondant. Dans ce cas, **DataGen** étend le schéma de base avec les contraintes de l'un des schémas spécifiés dans le tableau.

schema_inverter (Extension de schéma) : Si le schéma contient le mot-clé "not", cela signifie que l'instance ne doit pas valider le schéma spécifié. Dans ce cas, **DataGen** doit d'abord "inverser" le schéma spécifié pour obtenir un schéma complémentaire/opposé, puis étendre le schéma de base avec ce schéma complémentaire/opposé.

4.1.3 Étape 2 : *DSL*

La structure intermédiaire est ensuite utilisée pour générer un modèle DSL, qui est utilisé pour générer des instances de schémas. Le modèle DSL est une représentation plus abstraite du schéma JSON, qui est plus facile à manipuler et à en générer des données.

4.1.4 Étape 3 : *Dataset*

Le dataset est généré à partir du modèle DSL, qui est créé à partir de la structure intermédiaire. Comme déjà décrit, le programme **DataGen** est capable de générer plusieurs instances de datasets à partir d'un seul schéma, en randomisant les valeurs à chaque tentative.

4.1.5 Étape 4 : *Instance finale*

La traduction du dataset en XML ou JSON se fait à l'aide de la bibliothèque de sérialisation JSON/XML de **DataGen**. Cette bibliothèque est capable de prendre en entrée un modèle DSL ou un dataset généré et de le convertir en un fichier JSON ou XML valide.

Il est important de noter que la bibliothèque de sérialisation est capable de gérer des schémas JSON ou XML complexes, y compris ceux qui utilisent des mécanismes tels que le regroupement ou la récursivité. Cela permet à **DataGen** de générer des fichiers de sortie qui sont conformes aux spécifications du schéma d'entrée.

4.2 Fonctionnement des types spécifiques

4.2.1 string

La fonction "parseStringType" effectue l'analyse d'un objet JSON afin de générer une représentation de chaîne de caractères en fonction des propriétés spécifiées dans cet objet. Cette fonction est utilisée pour traiter des schémas de données et générer des valeurs de chaînes de caractères correspondantes, en fonction des contraintes définies dans l'objet JSON.

Analyse des propriétés "pattern" et "notFormat" : La fonction commence par vérifier si la propriété "pattern" est présente dans l'objet JSON. Si c'est le cas, elle génère une représentation de modèle de chaîne de caractères en utilisant la valeur de la propriété "pattern". Ensuite, elle examine si les propriétés "notFormat" et "format" sont présentes dans l'objet JSON. Si "format" est présent et que sa valeur est incluse dans la liste "notFormat", la propriété "format" est supprimée de l'objet JSON.

Traitement des propriétés "format" : La fonction vérifie si la propriété "format" est présente dans l'objet JSON et effectue des actions spécifiques en fonction de sa valeur. Différentes valeurs possibles de "format" sont prises en compte, telles que "date-time", "date", "time", "duration", "regex", "email", "hostname", "ipv4", "ipv6", "uuid", "uri", "iri", "uri-reference" et "iri-reference". Pour chaque cas, une représentation de chaîne de caractères est générée conformément à la spécification du format.

Génération de chaînes de caractères aléatoires : Si aucune des propriétés "pattern" ou "format" n'est présente dans l'objet JSON, la fonction génère une représentation de chaîne de caractères aléatoire. Elle utilise les propriétés "minLength" et "maxLength" pour définir la longueur minimale et maximale de la chaîne de caractères. Si "minLength" n'est pas spécifié, une longueur aléatoire entre 0 et 100 est utilisée comme valeur par défaut. La fonction garantit que "max" n'est pas inférieur à "min" en ajustant "max" pour être égal à "min + 100" si nécessaire. Conclusion : La fonction "parseStringType" fournit un mécanisme efficace pour analyser les contraintes spécifiées dans un objet JSON et générer des valeurs

de chaînes de caractères appropriées en fonction de ces contraintes. Cette approche est utile pour le traitement des schémas de données et peut être intégrée dans des applications nécessitant la génération dynamique de données conformes à des contraintes spécifiques.

4.2.2 number, integer

La fonction "parseNumericType" traite un objet JSON contenant des propriétés spécifiques pour les contraintes numériques. Elle génère ensuite une représentation de valeur numérique en fonction de ces contraintes.

Analyse des propriétés : La fonction extrait les propriétés "multipleOf", "notMultipleOf", "minimum", "maximum", "exclusiveMinimum" et "exclusiveMaximum" de l'objet JSON pour définir les contraintes numériques à appliquer à la représentation de valeur numérique. Les propriétés "integer" et "notInteger" sont également extraites pour déterminer si la valeur numérique générée doit être un entier ou un nombre à virgule flottante.

Traitement des contraintes "multipleOf" : Si la propriété "multipleOf" n'est pas définie, la fonction utilise une valeur par défaut "1" pour les nombres à virgule flottante et "0.43" pour les nombres à virgule fixe (non entiers). Si "integer" est défini, la valeur "1" est ajoutée à la liste des valeurs "multipleOf" pour garantir que la valeur générée sera un multiple entier de la valeur de "multipleOf".

Gestion des contraintes "notMultipleOf" : Si la propriété "notMultipleOf" est définie, la fonction filtre les valeurs "multipleOf" en éliminant celles qui figurent dans la liste "notMultipleOf". Si la liste "multipleOf" est vide après la filtration, la fonction utilise une valeur par défaut "1" pour les nombres à virgule flottante et "0.43" pour les nombres à virgule fixe.

Détermination des bornes : Les bornes supérieure ("max") et inférieure ("min") sont déterminées en fonction des propriétés "maximum", "exclusiveMaximum", "minimum" et "exclusiveMinimum". Si les bornes exclusives sont spécifiées, une petite valeur est ajoutée ou soustraite en fonction de la présence de décimales ("any_frac") pour éviter les bords de l'intervalle.

Gestion des cas sans contraintes : Si l'objet JSON ne contient aucune contrainte numérique ou seulement la propriété "integer", la fonction génère une valeur numérique aléatoire entre -1000 et 1000 (inclus) pour les entiers et les nombres à virgule flottante.

Génération de multiples aléatoires : Si aucune contrainte de plage n'est spécifiée, la fonction génère un multiple aléatoire de la valeur "multipleOf" (ou de la valeur par défaut "1" si elle n'est pas définie). La fonction gère différents scénarios en fonction de la présence de contraintes spécifiques, telles que "notMultipleOf" ou "notInteger".

Génération de valeurs numériques dans une plage spécifique : Si des contraintes de plage sont spécifiées, la fonction détermine tous les multiples possibles de la valeur "multipleOf" qui se trouvent dans l'intervalle défini par les bornes "max" et "min". Ensuite, elle filtre ces multiples en éliminant ceux qui figurent dans la liste "notMultipleOf" (le cas échéant) et génère un multiple aléatoire de la valeur "multipleOf" qui satisfait les contraintes de plage spécifiées.

4.3 Fonctionnement des objets

La fonction "parseObjectType" effectue le parsing d'un objet (type "object") à partir des propriétés spécifiées dans un objet JSON. Voici une explication détaillée de cette fonction pour un rapport scientifique :

Pré-traitement : La fonction commence par appeler "parseNotObjectKeys(json)" pour éliminer les clés qui ne sont pas spécifiques à un objet dans l'objet JSON.

Initialisation : Un nouvel objet vide "obj" est créé pour stocker les propriétés de l'objet final. La fonction détermine également le nombre de propriétés "required" et les limites de la taille de l'objet "minProps", "maxProps", et "size" à partir des propriétés spécifiées dans l'objet JSON.

Traitement des propriétés "required" : La fonction génère les propriétés "required" de l'objet final en utilisant la fonction "addProperties". Les propriétés "required" sont déterminées aléatoirement en utilisant "rand" pour sélectionner des propriétés parmi celles spécifiées dans l'objet JSON.

Traitement récursif des propriétés : Si une propriété "recursive" est spécifiée dans l'objet JSON, la fonction effectue un traitement récursif des propriétés en utilisant "json.recursive.prop" comme propriété à ajouter dans l'objet final.

Traitement des propriétés dépendantes : Si des propriétés dépendantes ("dependentRequired") sont spécifiées dans l'objet JSON, la fonction ajoute ces propriétés à l'objet final en utilisant la fonction "add-Properties". Les propriétés dépendantes sont déterminées en fonction des propriétés existantes dans l'objet final.

Traitement des propriétés "patternProperties" : Si des propriétés "patternProperties" sont spécifiées dans l'objet JSON, la fonction génère une propriété aléatoire en utilisant la fonction "RandExp" pour appliquer le schéma spécifié.

Traitement des propriétés supplémentaires : Si les propriétés "additionalProperties" ou "unevaluatedProperties" sont spécifiées dans l'objet JSON, la fonction ajoute des propriétés aléatoires non requises à l'objet final en utilisant la fonction "nonRequiredtext_randomProps".

Conversion de l'objet en chaîne : Une fois toutes les propriétés ajoutées, l'objet final est converti en une chaîne de caractères de la DSL (Domain-Specific Language) en utilisant la variable "str". Les propriétés de l'objet sont concaténées avec la chaîne "str" pour former la représentation finale de l'objet.

Gestion des cas vides : Si l'objet final est vide, la fonction ajoute une propriété "DFJS_EMPTY_OBJECT" pour représenter un objet vide.

Retour de l'objet : La chaîne "str" contenant la représentation de l'objet final est ensuite renvoyée par la fonction "parseObjectType".

4.4 Fonctionnement des arrays

La fonction "parseArrayType" effectue le parsing d'un tableau (type "array") à partir des propriétés spécifiées dans un objet JSON. Voici une explication détaillée de cette fonction pour un rapport scientifique :

Pré-traitement : La fonction commence par appeler "parseNotArrayKeys(json)" pour éliminer les clés qui ne sont pas spécifiques à un tableau dans l'objet JSON.

Initialisation : Un tableau vide "arr" est créé pour stocker les éléments du tableau final. La fonction détermine également le nombre d'éléments préfixés "prefixed" et si les éléments supplémentaires sont autorisés "additionalItems" à partir des propriétés spécifiées dans l'objet JSON.

Génération des éléments préfixés : La fonction génère les éléments préfixés du tableau en utilisant la fonction "parseJSON" pour chaque élément spécifié dans "json.prefixItems".

Génération des éléments "contains" : Si la propriété "contains" est spécifiée dans l'objet JSON et les éléments supplémentaires sont autorisés, la fonction génère des éléments du tableau en fonction des propriétés spécifiées dans "json.contains". Les éléments du tableau sont générés en fonction des contraintes "minContains" et "maxContains".

Génération des éléments restants : La fonction génère les éléments restants du tableau en utilisant la fonction "parseJSON" pour l'objet JSON "nonPrefixedSchema", qui représente le schéma réel des éléments du tableau. Les éléments restants sont générés jusqu'à atteindre la taille du tableau "arrLen.len".

Conversion du tableau en chaîne : Une fois tous les éléments ajoutés, le tableau final est converti en une chaîne de caractères de la DSL (Domain-Specific Language) pour représenter le tableau final.

Gestion des doublons : Si la propriété "uniqueItems" est spécifiée dans l'objet JSON et que certains éléments du tableau contiennent des valeurs uniques, la fonction génère une nouvelle fonction pour garantir que les éléments du tableau sont uniques. Les éléments du tableau sont vérifiés jusqu'à 10 fois pour s'assurer qu'il n'y a pas de doublons.

Retour du tableau : La chaîne de caractères représentant le tableau final est ensuite renvoyée par la fonction "parseArrayType".

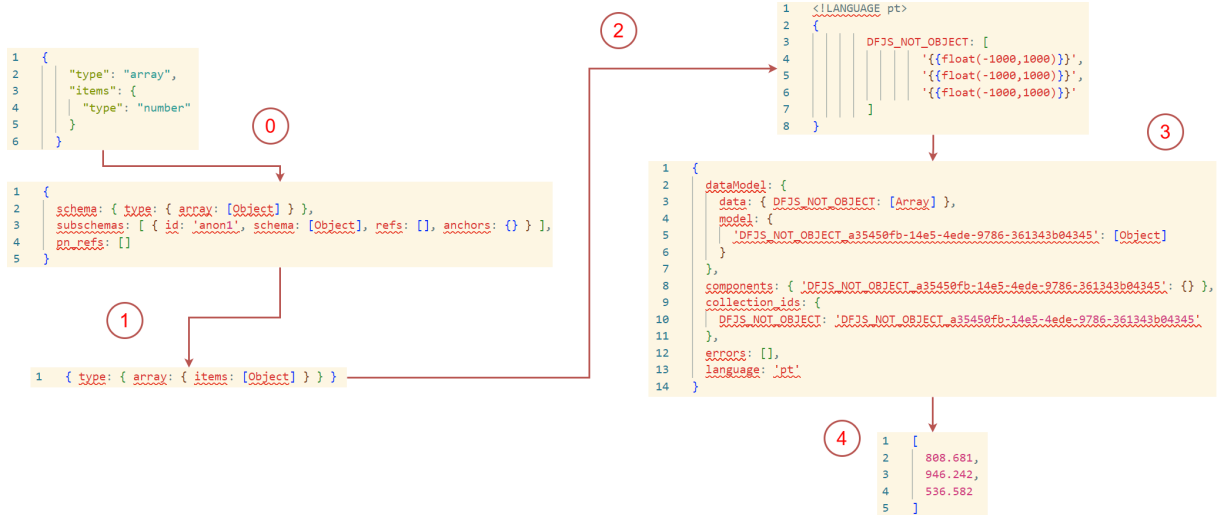


Figure 2 : Workflow d'un schema avec *array*

4.5 Fonctionnement des connecteurs logiques

4.5.1 Conjonction (*allOf*)

La fonction `parseAllSchemaComposition` parcourt le schéma JSON et détecte les clés de composition de schémas, y compris "allOf". Lorsqu'elle trouve "allOf", elle appelle la fonction `parseSchemaComposition` avec la clé "allOf" pour gérer cette composition spécifique.

La fonction `parseSchemaComposition` est appelée avec la clé "allOf" détectée. Elle récupère les sous-schémas de cette composition et les stocke dans la variable `subschemas`. Dans ce cas, les sous-schémas de "allOf" sont simplement les valeurs de la propriété "allOf" du schéma JSON initial.

La variable `subschemas` contient maintenant une liste de sous-schémas spécifiés dans "allOf". Ensuite, la fonction `parseAllSchemaComposition` est rappelée pour chaque sous-schéma trouvé dans `subschemas`, en utilisant la récursivité. Cela permet de traiter les compositions imbriquées à l'intérieur de "allOf".

Enfin, pour chaque sous-schéma dans `subschemas`, la fonction `extendSchema` est appelée. Cette fonction étend le schéma JSON initial avec les règles et propriétés spécifiées dans chaque sous-schéma.

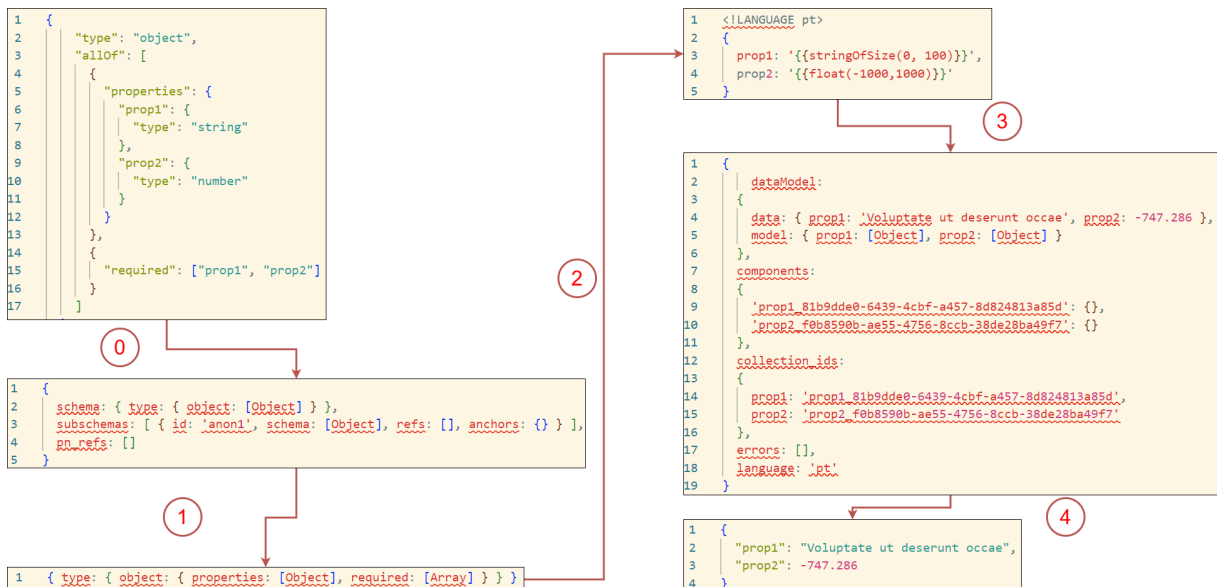


Figure 3 : Workflow d'un schema avec *allOf*

4.5.2 Disjonction (*oneOf*)

La fonction `parseAllSchemaComposition` parcourt le schéma JSON et détecte les clés de composition de schémas, y compris "oneOf". Lorsqu'elle trouve "oneOf", elle appelle la fonction `parseSchemaComposition` avec la clé "oneOf" pour gérer cette composition spécifique.

La fonction `parseSchemaComposition` est appelée avec la clé "oneOf" détectée. Elle récupère les sous-schémas de cette composition et les stocke dans la variable `subschemas`. Dans ce cas, les sous-schémas de "oneOf" sont choisis aléatoirement parmi les valeurs de la propriété "oneOf" du schéma JSON initial.

La variable `subschemas` contient maintenant une liste de sous-schémas choisis aléatoirement à partir de "oneOf". Ensuite, la fonction `parseAllSchemaComposition` est rappelée pour chaque sous-schéma trouvé dans `subschemas`, en utilisant la récursivité. Cela permet de traiter les compositions imbriquées à l'intérieur de "oneOf".

Enfin, pour chaque sous-schéma dans `subschemas`, la fonction `extendSchema` est appelée. Cette fonction étend le schéma JSON initial avec les règles et propriétés spécifiées dans chaque sous-schéma.

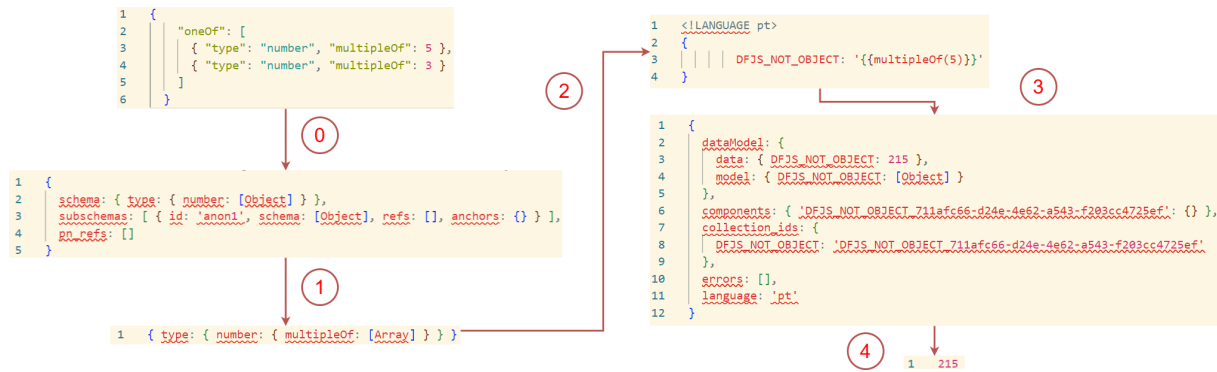


Figure 4 : Workflow d'un schema avec *oneOf*

4.5.3 Disjonction (*anyOf*)

La fonction `parseAllSchemaComposition` parcourt le schéma JSON et détecte les clés de composition de schémas, y compris "anyOf". Lorsqu'elle trouve "anyOf", elle appelle la fonction `parseSchemaComposition` avec la clé "anyOf" pour gérer cette composition spécifique.

La fonction `parseSchemaComposition` est appelée avec la clé "anyOf" détectée. Elle récupère les sous-schémas de cette composition et les stocke dans la variable `subschemas`. Dans ce cas, les sous-schémas de "anyOf" sont choisis aléatoirement à partir des valeurs de la propriété "anyOf" du schéma JSON initial.

La variable `subschemas` contient maintenant une liste de sous-schémas choisis aléatoirement à partir de "anyOf". Ensuite, la fonction `parseAllSchemaComposition` est rappelée pour chaque sous-schéma trouvé dans `subschemas`, en utilisant la récursivité. Cela permet de traiter les compositions imbriquées à l'intérieur de "anyOf".

Enfin, pour chaque sous-schéma dans `subschemas`, la fonction `extendSchema` est appelée. Cette fonction étend le schéma JSON initial avec les règles et propriétés spécifiées dans chaque sous-schéma.

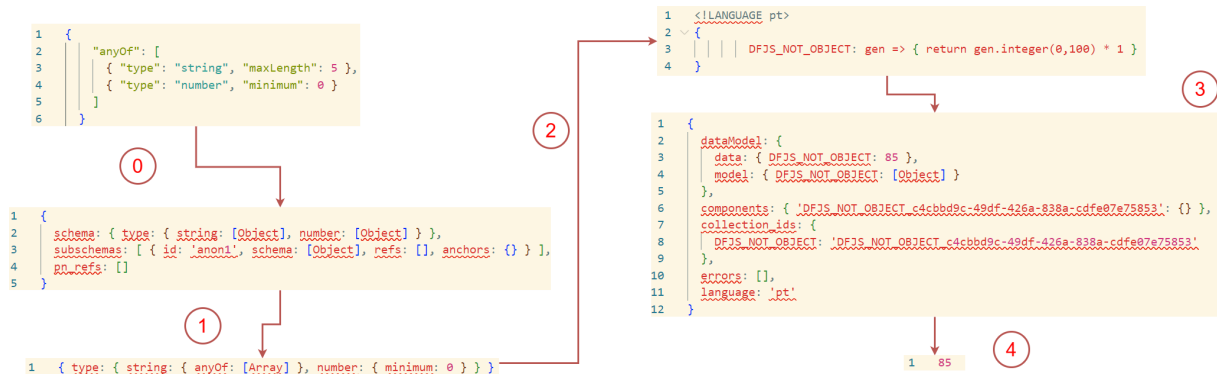


Figure 5 : Workflow d'un schema avec *anyOf*

4.5.4 Négation (*not*)

La fonction `parseAllSchemaComposition` parcourt le schéma JSON et détecte les clés de composition de schémas, y compris "not". Lorsqu'elle trouve "not", elle appelle la fonction `parseSchemaComposition` avec la clé "not" pour gérer cette composition spécifique.

La fonction `parseSchemaComposition` est appelée avec la clé "not" détectée. Elle récupère le sous-schéma de cette composition et le stocke dans la variable `subschemas`. Dans ce cas, le sous-schéma de "not" est simplement la valeur de la propriété "not" du schéma JSON initial.

La variable `subschemas` contient maintenant le sous-schéma spécifié dans "not". Ensuite, la fonction `parseAllSchemaComposition` est rappelée pour le sous-schéma trouvé dans `subschemas`, en utilisant la récursivité. Cela permet de traiter les compositions imbriquées à l'intérieur de "not".

Pour le sous-schéma dans `subschemas`, la fonction `extendSchema` est appelée. et qui en détectent la présence du `not` lance la fonction d'inversion.

La fonction utilise l'objet `invert` pour inverser les clés spécifiées dans le sous-schéma. Par exemple, si le sous-schéma contient une clé "properties" qui indique quelles propriétés sont autorisées, cette clé sera inversée pour devenir "notProperties", indiquant quelles propriétés ne sont pas autorisées.

La fonction utilise ensuite une instruction `switch` pour traiter le type spécifique du schéma JSON. Selon le type ("number", "string", "object" ou "array"), la fonction applique des règles spécifiques pour inverser certaines propriétés du schéma. Par exemple, pour le type "number", elle appelle la fonction `invert.notNumeric(schema)` pour inverser les règles spécifiques aux nombres, ce qui est équivalent à accepter tout type sauf une "number"

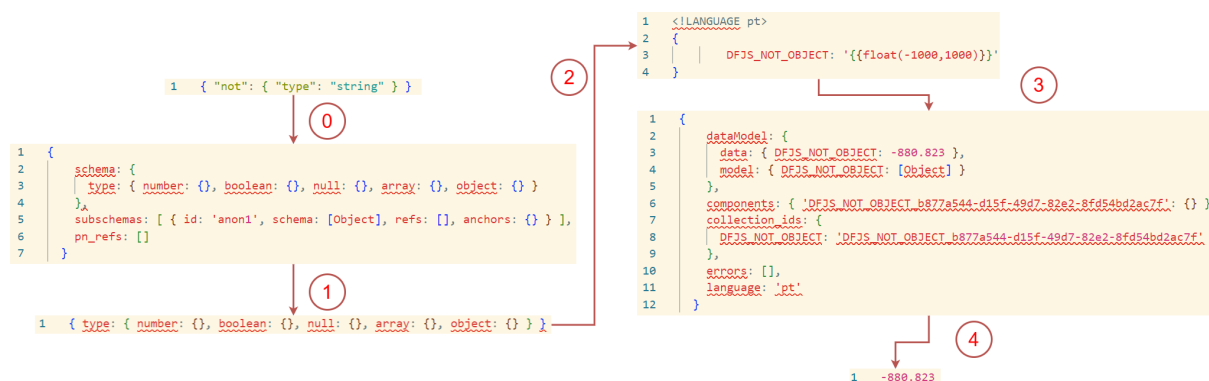


Figure 6 : Workflow d'un schema avec *not*

4.6 Explication du fonctionnement de l'extension de schéma

Importation du module `invert` : Le code importe un module nommé `invert` à partir du fichier `schema_invert.js`. Le module contient des fonctions pour inverser certaines propriétés du schéma JSON, permettant ainsi d'exprimer des règles de négation dans le schéma final.

Fonction `extendSchema` : Cette fonction est au cœur du processus d'extension du schéma JSON. Elle prend en entrée un schéma JSON initial (`json`), un sous-schéma spécifique à appliquer (`schema`), le type du schéma initial (`type`), la clé de composition (`key`) et des paramètres de configuration (`SETTINGS`).

Traitement de la composition "not" : Si la clé de composition (`key`) est égale à "not", cela signifie que le sous-schéma fourni est une composition de négation. Dans ce cas, certaines propriétés du schéma initial doivent être inversées. Le module `invert` est utilisé pour réaliser cette inversion.

Extension des propriétés spécifiques : La fonction `extendSchema` appelle différentes sous-fonctions (`extendArrayKey`, `extendNumeric`, `extendString`, `extendObject`, `extendArray`, `extendSizeKeys`, `assignProperties`, `assignSchemaObject`, `assignSubschema`) pour gérer spécifiquement chaque type de propriété du schéma JSON.

Gestion des propriétés de tableau (`extendArrayKey`) : Cette fonction étend les propriétés de type tableau telles que "const", "enum", "default", "notValues" et "notDefault" en combinant les valeurs du sous-schéma avec celles du schéma initial.

Gestion des propriétés de type chaîne (extendString) : Cette fonction étend les propriétés spécifiques des schémas de type chaîne tels que "pattern", "format", et "notFormat".

Gestion des propriétés de type numérique (extendNumeric) : Cette fonction gère l'extension des propriétés spécifiques aux schémas de type numérique comme "minimum", "maximum", "multipleOf", etc., en tenant compte des valeurs minimales et maximales.

Gestion des propriétés de type objet (extendObject) : Cette fonction traite l'extension des propriétés spécifiques aux schémas de type objet, telles que "properties", "patternProperties", "additionalProperties", etc.

Gestion des propriétés de type tableau (extendArray) : Cette fonction gère l'extension des propriétés spécifiques aux schémas de type tableau, comme "items", "contains", "uniqueItems", etc.

Gestion des clés de taille (extendSizeKeys) : Cette fonction gère l'extension des clés de taille, comme "minLength", "maxLength", "minItems", "maxItems", "minProperties" et "maxProperties".

Gestion des propriétés de type objet (assignProperties) : Cette fonction gère l'attribution des propriétés de type objet, telles que "properties" et "patternProperties", en tenant compte des règles de fusion définies par les paramètres de configuration SETTINGS.

Gestion des propriétés de type objet (assignSchemaObject) : Cette fonction gère l'attribution des propriétés de type objet, telles que "additionalProperties", "unevaluatedProperties" et "propertyNames", en tenant compte des règles de fusion définies par les paramètres de configuration SETTINGS.

Gestion des sous-schémas (assignSubschema) : Cette fonction gère l'attribution des sous-schémas spécifiques aux types JSON spécifiés. Elle permet de gérer les schémas imbriqués et de composer les règles spécifiques pour chaque type JSON.

4.7 Explication du fonctionnement de la négation de schéma

Fonction notSubschema : Cette fonction prend un schéma JSON (json) en entrée et traite les sous-schémas spécifiés pour l'opération de négation. Elle itère à travers les différents types spécifiés dans le schéma et gère la conversion du schéma en son schéma de négation correspondant.

Traitement des types du schéma : La fonction parcourt les différents types définis dans le schéma JSON. Si un type est vide (ne contient aucune contrainte), il est ajouté à la liste des types à exclure (notTypes) et supprimé du schéma initial.

Traitement des propriétés génériques (notGenericKeys) : Certaines propriétés génériques comme "const", "enum", et "default" doivent être transformées en leur équivalent pour le schéma de négation. La fonction notGenericKeys effectue cette transformation en renommant les propriétés et en les déplaçant dans le schéma de négation.

Traitement des propriétés numériques (notNumeric) : Cette fonction gère les propriétés spécifiques aux schémas numériques pour le schéma de négation. Elle inverse les contraintes "minimum", "maximum", "exclusiveMinimum", "exclusiveMaximum", "multipleOf" et "notMultipleOf".

Traitement des propriétés de type chaîne (notString) : Pour les schémas de type chaîne, la fonction gère les propriétés spécifiques telles que "pattern", "format", "minLength" et "maxLength", et les transforme en leurs équivalents pour le schéma de négation.

Traitement des propriétés de type objet (notObject) : Pour les schémas de type objet, la fonction gère les propriétés spécifiques telles que "properties", "patternProperties", "additionalProperties", "unevaluatedProperties", et "required" pour le schéma de négation.

Traitement des propriétés de type tableau (notArray) : Pour les schémas de type tableau, la fonction gère les propriétés spécifiques telles que "items", "unevaluatedItems", "minItems", "maxItems", "uniqueItems" et "contains" pour le schéma de négation.

Gestion des clés de taille (notSizeKeys) : Cette fonction gère l'inversion des clés de taille, comme "minLength" et "maxLength", "minItems" et "maxItems", en les transformant en contraintes d'exclusion.

Gestion des propriétés de type objet (notProperties) : Cette fonction gère l'inversion des sous-schémas pour les propriétés de type objet, telles que "properties" et "patternProperties".

Gestion des autres éléments (`notOtherElements`) : Cette fonction gère l'inversion des propriétés de type objet pour les propriétés autres que "properties" et "patternProperties", comme "additionalProperties" et "unevaluatedProperties".

4.8 Explication du modèle DSL

Le DSL (Domain Specific Language) de **DataGen** est un langage de spécification de données qui permet à l'utilisateur de décrire la structure et la sémantique des ensembles de données à générer. Le DSL est conçu pour être facile à utiliser et à comprendre, même pour les utilisateurs qui ne sont pas des experts en programmation.

Le DSL de **DataGen** permet à l'utilisateur de spécifier différents types de données, des relations locales entre les champs, l'utilisation de données provenant de jeux de données de support décrivant différentes catégories, la hiérarchie structurelle de l'ensemble de données et de nombreuses autres propriétés pertinentes. Le DSL est doté d'une large gamme de fonctionnalités qui permettent à l'utilisateur de spécifier des ensembles de données très variés et représentatifs en JSON ou XML, tout en traitant des exigences complexes et exigeantes.

Le DSL de **DataGen** est basé sur une grammaire JSON Schema, qui permet à l'utilisateur de spécifier la structure et la sémantique des ensembles de données de manière claire et concise. La grammaire JSON Schema est une norme de l'industrie pour la spécification de schémas JSON, ce qui facilite l'apprentissage et l'utilisation du DSL de **DataGen**.

4.9 Explication du modèle Dataset pré XML/JSON

Le dataset généré par **DataGen** avant sa conversion en JSON ou XML est un ensemble de données synthétiques qui est créé en fonction des spécifications fournies par l'utilisateur dans le DSL de **DataGen**. Le dataset est généré en temps d'exécution, ce qui signifie que les valeurs sont créées spontanément à partir d'une bibliothèque de jeux de données de support ou en utilisant des mécanismes de génération flous.

Le dataset généré par **DataGen** est structuré en fonction des spécifications de l'utilisateur, qui peuvent inclure des relations locales entre les champs, l'utilisation de données provenant de jeux de données de support décrivant différentes catégories, la hiérarchie structurelle de l'ensemble de données et de nombreuses autres propriétés pertinentes. Le dataset peut également inclure des mécanismes de répétition et de génération flous, qui permettent de générer des ensembles de données plus variés et représentatifs.

Le dataset généré par **DataGen** est conçu pour être réaliste et représentatif des données réelles, tout en respectant les restrictions structurelles et sémantiques spécifiées par l'utilisateur. Le dataset peut être utilisé pour une variété de tâches, telles que le test de logiciels, la recherche scientifique et l'analyse de données.

5 Étude des limitations de DataGen

5.1 Choix de la librairie

En ce qui concerne **json-schema-faker**, cette bibliothèque avait déjà fait l'objet d'une étude préalable documenté [7], nous avons donc suivis le même processus pour l'étude du **DataGen**

DataGen From Schemas est une extension de la version précédente de **DataGen**, qui permet de générer des ensembles de données synthétiques directement à partir de schémas JSON et XML. Les deux plates-formes partagent la même couche de données, ce qui assure la compatibilité des deux plates-formes et la portabilité des modèles DSL créés entre elles.

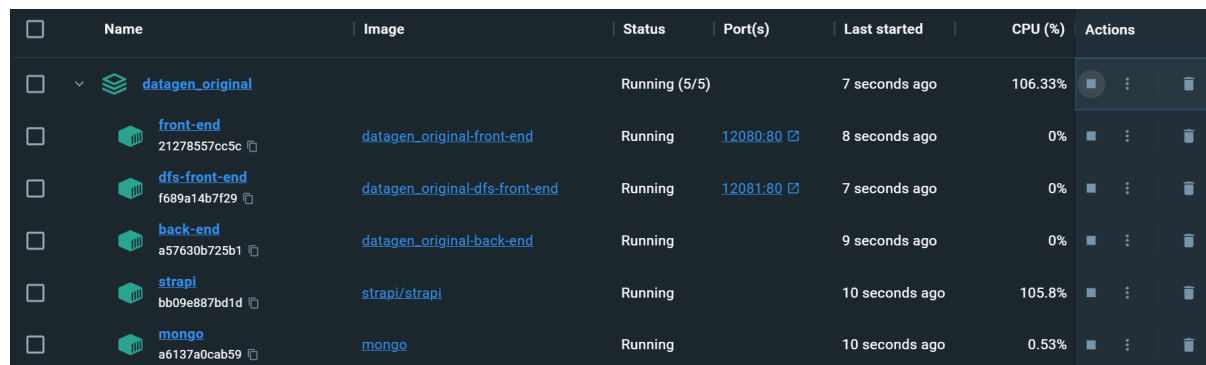
DataGen From Schemas est conçu pour étendre l'utilisation de **DataGen** en augmentant le niveau d'abstraction de la spécification des ensembles de données. Cette nouvelle plateforme permet de générer des ensembles de données synthétiques directement à partir de schémas, ce qui facilite la tâche de spécification des ensembles de données et permet de générer des ensembles de données plus rapidement et plus facilement.

En outre, **DataGen From Schemas** utilise la même grammaire JSON Schema que la version précédente de **DataGen**, ce qui permet une transition en douceur entre les deux plates-formes. Les utilisateurs

qui sont familiers avec la version précédente de **DataGen** peuvent facilement passer à **DataGen From Schemas** sans avoir à apprendre une nouvelle grammaire ou une nouvelle syntaxe.

5.1.1 Prise en main et installation du générateur DataGen

DataGen From Schemas est intégré dans l'application d'origine, car les deux composants fonctionnent de concert et sont encapsulés par la même instance Docker. Le processus d'installation de l'application est assez simple



Name	Image	Status	Port(s)	Last started	CPU (%)	Actions
datagen_original		Running (5/5)		7 seconds ago	106.33%	[Stop] [Refresh] [Delete]
front-end 21278557cc5c	datagen_original-front-end	Running	12080:80	8 seconds ago	0%	[Stop] [Refresh] [Delete]
dfs-front-end f689a14b7f29	datagen_original-dfs-front-end	Running	12081:80	7 seconds ago	0%	[Stop] [Refresh] [Delete]
back-end a57630b725b1	datagen_original-back-end	Running		9 seconds ago	0%	[Stop] [Refresh] [Delete]
strapi bb09e887bd1d	strapi/strapi	Running		10 seconds ago	105.8%	[Stop] [Refresh] [Delete]
mongo a6137a0cab59	mongo	Running		10 seconds ago	0.53%	[Stop] [Refresh] [Delete]

Figure 7 : Container du DataGen et ses images

L'utilisation de Docker dans le contexte de **DataGen** permet d'offrir un environnement de développement et d'exécution cohérent et isolé. Docker est une technologie de virtualisation légère basée sur des conteneurs, qui permet de créer des environnements d'exécution autonomes pour les applications. Cette approche s'inscrit dans une démarche de virtualisation au niveau du système d'exploitation, où chaque application et ses dépendances sont encapsulées dans un conteneur indépendant. Lorsque nous intégrons **DataGen** dans l'application d'origine à l'aide de Docker, plusieurs avantages se présentent :

Isolation : Les conteneurs Docker fournissent une isolation complète des dépendances et des bibliothèques de **DataGen** par rapport au reste de l'application. Ainsi, les conflits potentiels avec d'autres composants de l'application sont évités, et les dépendances spécifiques à **DataGen** sont gérées de manière indépendante.

Portabilité : Grâce à Docker, l'environnement de développement et d'exécution est le même sur n'importe quelle machine qui exécute Docker. Cela permet une meilleure portabilité de l'application, car les mêmes conteneurs peuvent être utilisés pour le développement, les tests et la production, sans se soucier des différences entre les systèmes d'exploitation ou les configurations matérielles.

Répétabilité : Avec Docker, les images de conteneurs sont définies par des fichiers de configuration appelés Dockerfiles. Ces fichiers décrivent de manière reproductible tous les composants et dépendances nécessaires à **DataGen**. Ainsi, l'ensemble du processus de construction et de déploiement peut être reproduit de manière fiable sur différentes machines.

Facilité de gestion : Docker fournit des outils de gestion puissants pour créer, déployer, mettre à jour et surveiller les conteneurs. Cela simplifie considérablement le déploiement et la gestion de **DataGen** au sein de l'application, rendant le processus plus efficace et moins sujet aux erreurs.

Évolutivité : L'utilisation de Docker facilite l'évolutivité de l'application, car les conteneurs peuvent être facilement mis à l'échelle horizontalement en fonction des besoins de charge et de performance.

5.2 Choix du validateur JSON Schema

Nous avons opté pour le validateur *json-schema-validator* [9] qui est une implémentation Java du standard JSON Schema Core Draft, prenant en charge les versions v4, v6, v7, v2019-09 et v2020-12 (partielle). Ce validateur utilise Jackson, le parseur JSON le plus populaire en Java, par défaut. La performance a été une priorité dans sa conception

Principales caractéristiques de cette bibliothèque :

Performance : L'implémentation est remarquablement rapide, environ 32 fois plus rapide que Fge et cinq fois plus rapide que Everit.

Parseur : En utilisant Jackson, il s'intègre naturellement aux projets déjà utilisant le parseur Jackson pour JSON.

Prise en charge YAML : Elle prend en charge JSON et YAML pour les définitions de schémas et les données d'entrée.

Dépendances minimales : La bibliothèque a été conçue avec un minimum de dépendances pour éviter les conflits.

5.3 Approche adoptée

L'approche adoptée se décompose en plusieurs étapes. Tout d'abord, nous vérifions la classification des erreurs générée grâce à un pipeline (que nous détaillons dans la section "Expérimentations" de ce rapport), afin de nous concentrer sur celles qui sont les plus fréquentes et d'avoir accès à leurs schémas associés.

Ensuite, nous prenons un ou plusieurs de ces schémas et les réduisons pour isoler la partie liée à l'erreur. À partir de là, nous montons une hypothèse sur l'erreur en nous appuyant également sur la sortie du validateur.

Nous confirmons ensuite cette hypothèse en utilisant le code du générateur à l'aide d'un débogueur, celui de l'IDE (*WebStorm 2022.3.2*). Enfin, nous avons la possibilité d'essayer de trouver une correction possible au niveau du code, ce qui implique de relancer le générateur et le validateur pour confirmer l'efficacité de la correction.

En somme, cette approche adoptée permet une analyse systématique des erreurs, une compréhension approfondie de leurs causes et une résolution précise dans les cas où cela a été possible.

5.4 Expérimentations

5.4.1 Étapes suivies

La première étape consiste à faire la traduction des schémas des différents datasets à partir d'un quelconque draft vers celui 2020-12, vu que l'outil **DataGen** implémente uniquement la syntaxe 2020-12

Une fois que nous avons fait la traduction, la deuxième étape est la génération de masse d'instances JSON Schema pour un ensemble de datasets de JSON Schema. En simultanée sera faite la collecte de erreurs de génération fournie par le **DataGen**.

La troisième étape a consisté à valider ces instances par rapport aux schémas JSON correspondants. Nous avons utilisé un validateur d'instances JSON Schema, l'outil *json-schema-validator*.

Enfin, la quatrième étape consiste à classer ces schémas selon l'erreur de validation de leurs instances générées. Cette classification nous permet d'analyser les schémas ayant des erreurs similaires ensemble, ce qui facilite l'analyse du code du générateur. Nous avons utilisé l'IDE WebStorm de la suite JetBrains pour cette étape.

Grâce à ces étapes et leurs différents script implémentés par nos soins, nous avons pu générer des données de manière automatique et efficace, et nous avons également pu identifier des erreurs dans les schémas JSON générés. Cela nous a permis d'analyser les schémas à problèmes et d'améliorer les générateurs open-source utilisés.

Une étape annexe qui a surtout servi à la compréhension du workflow fut l'ajout de logs au code pour suivre l'évolution d'un schéma en entré

```

back-end
datagen-back-end
9fd706ea63ec

Logs Inspect Terminal Files Stats

2023-08-04 04:58:18 schema parsed
2023-08-04 04:58:18 ETAPE 0: Schema original
2023-08-04 04:58:18 {
2023-08-04 04:58:18   schema: { type: { object: [Object] } },
2023-08-04 04:58:18   subschemas: [ { id: 'anon1', schema: [Object], refs: [], anchors: [] } ],
2023-08-04 04:58:18   pn_refs: []
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 1: Schema etendu
2023-08-04 04:58:18 ETAPE 1.1: Extension du schema (ops de composition)
2023-08-04 04:58:18 ETAPE 1.1: Detection du 'allOf'
2023-08-04 04:58:18 {
2023-08-04 04:58:18   properties: { prop1: [Object], prop2: [Object] },
2023-08-04 04:58:18   required: [ 'prop1', 'prop2' ]
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 1.3: Schema etendu, non clean
2023-08-04 04:58:18 {
2023-08-04 04:58:18   allOf: [ { properties: [Object] }, { required: [Array] } ],
2023-08-04 04:58:18   properties: { prop1: { type: [Object] }, prop2: { type: [Object] } },
2023-08-04 04:58:18   required: [ 'prop1', 'prop2' ]
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 1.4: Schema etendu, cleaned
2023-08-04 04:58:18 {
2023-08-04 04:58:18   properties: { prop1: { type: [Object] }, prop2: { type: [Object] } },
2023-08-04 04:58:18   required: [ 'prop1', 'prop2' ]
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 1.5: Schema etendu, final
2023-08-04 04:58:18 { type: { object: { properties: [Object], required: [Array] } } }
2023-08-04 04:58:18 ETAPE 2: Creation du DSL (parsing)
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types: cas 'object'
2023-08-04 04:58:18 ETAPE 2.2: Parsing des 'NotObjectKeys'
2023-08-04 04:58:18 ETAPE 1.1: Extension du schema (ops de composition)
2023-08-04 04:58:18 ETAPE 1.5: Schema etendu, final
2023-08-04 04:58:18 { type: { string: {} } }
2023-08-04 04:58:18 ETAPE 2: Creation du DSL (parsing)
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types: cas 'string'
2023-08-04 04:58:18 '{{stringOfSize(0, 100}})'
2023-08-04 04:58:18 ETAPE 1.1: Extension du schema (ops de composition)
2023-08-04 04:58:18 ETAPE 1.5: Schema etendu, final
2023-08-04 04:58:18 { type: { number: {} } }
2023-08-04 04:58:18 ETAPE 2: Creation du DSL (parsing)
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types
2023-08-04 04:58:18 ETAPE 2.1: Parsing des types: cas 'number'
2023-08-04 04:58:18 '{{float(-1000,1000}})'
2023-08-04 04:58:18 {
2023-08-04 04:58:18   prop1: '{{stringOfSize(0, 100}})',
2023-08-04 04:58:18   prop2: '{{float(-1000,1000}})'
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 2.2: DSL final
2023-08-04 04:58:18 -iLANGUAGE pt-
2023-08-04 04:58:18 {
2023-08-04 04:58:18   prop1: '{{stringOfSize(0, 100}})',
2023-08-04 04:58:18   prop2: '{{float(-1000,1000}})'
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 3: Dataset genere
2023-08-04 04:58:18 {
2023-08-04 04:58:18   dataModel: {
2023-08-04 04:58:18     data: {
2023-08-04 04:58:18       prop1: 'Ex non sunt mollit tempor labore sit. Fugiat voluptate occaecat cons',
2023-08-04 04:58:18       prop2: -886.442
2023-08-04 04:58:18     },
2023-08-04 04:58:18     model: { prop1: [Object], prop2: [Object] }
2023-08-04 04:58:18   },
2023-08-04 04:58:18   components: {
2023-08-04 04:58:18     'prop1_36bfff17-7d68-44d2-af82-1575bf88e173': {},
2023-08-04 04:58:18     'prop2_07685429-7c98-4c7f-b9d2-5d95647a428d': {}
2023-08-04 04:58:18   },
2023-08-04 04:58:18   collection_ids: {
2023-08-04 04:58:18     prop1: 'prop1_36bfff17-7d68-44d2-af82-1575bf88e173',
2023-08-04 04:58:18     prop2: 'prop2_07685429-7c98-4c7f-b9d2-5d95647a428d'
2023-08-04 04:58:18   },
2023-08-04 04:58:18   errors: [],
2023-08-04 04:58:18   language: 'pt'
2023-08-04 04:58:18 }
2023-08-04 04:58:18 ETAPE 4: Dataset converti en JSON
2023-08-04 04:58:18 {
2023-08-04 04:58:18   "prop1": "Ex non sunt mollit tempor labore sit. Fugiat voluptate occaecat cons",
2023-08-04 04:58:18   "prop2": -886.442
2023-08-04 04:58:18 }
2023-08-04 04:58:18 POST /api/json_schema/ 201 126.852 ms - 129

```

Figure 8 : Logs du l'image du Back-end du Container du DataGen

5.4.2 Collections de schémas utilisés

Tout comme l'analyse du `json-schema-faker`, nous avons choisi reutiliser quatre datasets (Snowplow [11], WashingtonPost [12], Kubernetes [13] et GitHub [14]), ajouter a cela deux autres datasets, Containment [15] et Handwritten pour un total de 6099 instances générées. Nous les avons repris d'un article publié. Le choix est aussi justifié par leur intégration dans des domaines applicatifs spécifiques, ce qui les rend plus représentatifs et pertinents pour les tests.

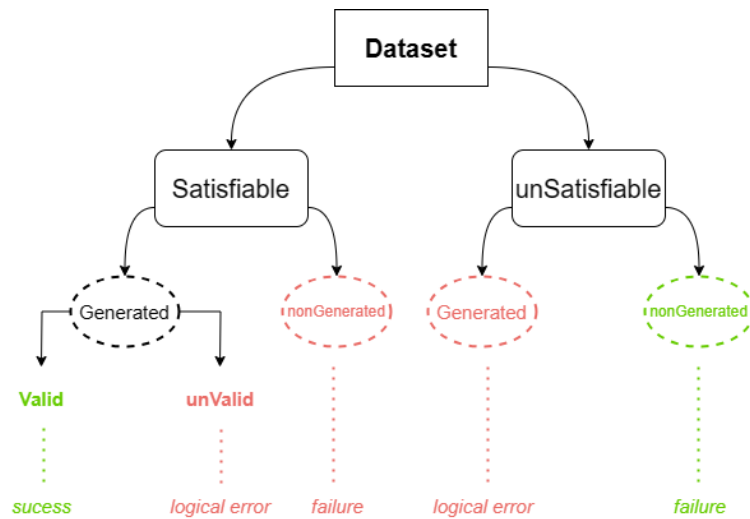
Parmi eux, certains sont classifiés comme des schémas satisfiables (sat), ce qui signifie qu'ils ont des instances valides possibles. D'autres sont classifiés comme insatisfiables (unsat), ce qui signifie qu'ils sont considérés comme illogiques car il n'existe pas d'instance valide possible pour les satisfaire.

Générateur	Dataset	Total Sat/unSat (schémas) (schémas)	Generated Sat/unSat (schémas) (schémas)	Taux de génération (%)	Instance Valid/Invalid parmis Generated Sat	Taux de validité parmis Generated Sat (%)
json-schema-faker	Snowplow	420 / 0	409 / 0	97.38 %	402 / 7	98.28 %
	Wp	125 / 0	125 / 0	100.00 %	102 / 23	81.60 %
	Kubernetes	1087 / 5	1082 / 0	99.08 %	984 / 98	90.94 %
	GitHub	6387 / 40	5957 / 0	92.69 %	5254 / 703	88.19 %
DataGen	Snowplow	420 / 0	385 / 0	91.67 %	351 / 34	91.16 %
	Wp	125 / 0	93 / 0	74.40 %	91 / 2	97.85 %
	Kubernetes	1087 / 5	862 / 4	79.30 %	724 / 138	84.00 %
	GitHub	6387 / 40	4450 / 3	69.29 %	3765 / 645	84.61 %
	Containment	450 / 881	31 / 154	13.90 %	3 / 28	09.68 %
	Handwritten	188 / 38	95 / 22	51.77 %	15 / 80	15.79 %

Tableau 1 : Taux de validité des instances générées

5.4.3 Classement des résultats

En vu des différents résultats obtenus, on peut distinguer 5 cas :



5.5 Remarques

Notre travail sur les limitations de **DataGen** et notre analyse de son code nous ont permis de faire une observation principale : **DataGen** traite les opérateurs (**oneOf**, **allOf**, **not**, etc.) de manière singulière, mais éprouve des difficultés lorsqu'un schéma contient une composition et une imbrication de ces opérateurs.

6 Contraintes

6.1 Linguistique

La totalité des commentaires du code source du **DataGen** est rédigée en portugais, tandis qu'une partie des documents le sont également. Cette situation peut présenter un obstacle linguistique pour l'équipe vu qu'aucun ne maîtrisent pas le portugais, rendant l'interprétation du code et de certaines parties de la documentation plus difficile.

6.2 Flux de traitement intégré

DataGen From Schemas s'exécute en tant qu'extension de **DataGen** et traite le schéma spécifié avant que les étapes de génération de données du **DataGen** original ne soient appliquées. Cela conduit à une chaîne de traitement complexe comprenant de multiples étapes provenant des deux modules, ce qui peut rendre la compréhension et le suivi du processus plus difficiles en raison de la complexité accrue."

6.3 Matériel

Le processus de lancement du conteneur "datagen" et ses cinq images associées consomme une quantité excessive de ressources système, entraînant une charge élevée sur le système hôte. Cela se traduit par une utilisation élevée de la mémoire, du CPU et d'autres ressources, entraînant des temps de démarrage plus longs et des performances globales réduites.

6.4 Logiciel

Une contrainte logicielle a été rencontrée, indiquant que l'entité de la requête est trop volumineuse ('request entity too large'). Les informations associées à cette contrainte sont les suivantes : 'expected' : 412184, 'length' : 412184, 'limit' : 102400, 'type' : 'entity.too.large'. Cela signifie que la taille de l'entité de la requête attendue était de 412184 octets, mais la taille réelle de l'entité reçue était également de 412184 octets, dépassant ainsi la limite autorisée de 102400 octets ('limit' : 102400).

Pour résoudre de manière naïve le problème de taille des schémas JSON atteignant plusieurs dizaines de kilo-octets, nous avons simplement ignoré ces schémas volumineux, limitant ainsi le traitement aux schémas plus petits.

Quant au problème logiciel 502 Bad Gateway, pour le résoudre également de manière naïve, nous avons choisi de fragmenter les jeux de données en morceaux plus petits avant la phase de génération de masse. Cela a permis d'éviter les erreurs 502 en répartissant la charge sur plusieurs requêtes plus petites. Cependant, cette approche naïve a ses limites car elle entraîne un ralentissement de l'automatisation de la phase de génération de masse, étant donné qu'elle nécessite une fragmentation manuelle des données en préparation.

6.5 Débogage

Une autre contrainte que nous avons rencontrée lors du projet est liée au débogage du **DataGen** sur WebStorm. Malgré de nombreuses tentatives d'adaptation du code et de configuration, nous n'avons pas pu résoudre le problème des instances vides qui apparaissaient de manière récurrente lors du débogage. Ce problème est probablement dû à un conflit de version JavaScript entre le code du **DataGen** et le débogueur de WebStorm.

Cette limitation a rendu le processus de débogage plus difficile, car nous n'avions pas accès aux informations détaillées sur les erreurs et les problèmes rencontrés par le code. Cela a pu ralentir notre capacité à identifier et à résoudre certains problèmes spécifiques.

7 Perspectives

Une des contraintes principales de notre projet a été le manque de temps, principalement dû à l'effort considérable consacré à la mise en place de l'environnement d'exécution et à la compréhension du code source, qui a été réalisée simultanément.

En effet, nous avons dû allouer une part importante de notre emploi du temps à configurer l'environnement nécessaire pour exécuter les outils de génération et de validation, ainsi qu'à résoudre les éventuels problèmes de dépendances et de compatibilité logicielle.

De plus, la compréhension du code source du projet a été un processus qui s'est déroulé en parallèle avec la configuration de l'environnement. Cette double tâche a nécessité un effort supplémentaire pour assimiler la logique du code, comprendre les interactions entre les différentes parties du système et décrypter les choix de conception réalisés par les développeurs.

Malheureusement, ce temps considérable investi dans la mise en place de l'environnement et la compréhension du code a eu un impact sur notre capacité à explorer certains aspects plus avancés de notre travail, tels que l'analyse approfondie des erreurs ou l'expérimentation avec d'autres générateurs open source.

Malgré cette contrainte de temps, nous avons tout de même réussi à réaliser une analyse initiale du projet et à obtenir des résultats significatifs. Nous reconnaissons que des opportunités d'approfondissement existent et que d'autres recherches pourraient être entreprises dans le futur pour tirer pleinement parti de ces résultats préliminaires.

8 Conclusion

Après avoir mené une étude approfondie des générateurs open-source, nous entrepris de caractériser les limitations d'un générateur spécifique en termes de classes de schémas JSON correctement traitées et de classes de schémas présentant des problèmes. notre approche a été basée sur des méthodes de rétro-ingénierie et des analyses expérimentales, nous permettant d'évaluer l'efficacité ainsi que les limites de ce générateur.

L'analyse minutieuse des résultats nous a permis de mieux comprendre les classes de schémas qui posent des difficultés aux générateurs open-source, tout en identifiant les domaines dans lesquels ce générateur pourrait bénéficier d'améliorations. On a constaté que les générateurs open-source actuels peuvent traiter efficacement de nombreux types de schémas JSON, mais qu'ils peuvent rencontrer des difficultés lorsqu'ils sont confrontés à des schémas plus complexes et interconnectés.

J'espère sincèrement que ce projet pourra apporter des connaissances précieuses aux développeurs de générateurs open-source ainsi qu'aux utilisateurs de ces outils. En mettant en évidence les limites actuelles des générateurs et en identifiant les domaines où des améliorations sont nécessaires, j'aspire à contribuer à l'amélioration de la qualité et des fonctionnalités des générateurs open-source. Ces avancées bénéficieront à toute la communauté des développeurs et des utilisateurs de logiciels.

J'ai eu le privilège de travailler avec d'excellents chercheurs au sein de ce laboratoire. Leur expertise, leur passion pour la recherche et leur soutien ont été des atouts inestimables pour mener à bien ce projet. Leur savoir-faire m'a permis d'apprendre énormément et de bénéficier d'un environnement stimulant propice à l'épanouissement de mes compétences. Travailler aux côtés de ces chercheurs talentueux a été une expérience enrichissante, et je suis reconnaissant de l'opportunité qui m'a été offerte de collaborer avec eux au sein de l'équipe BD du LIP6.

Ce stage a été une occasion unique pour moi d'acquérir de l'expérience et d'apporter ma contribution. J'espère que les résultats de notre travail seront utiles aux futurs projets de recherche de l'équipe.

Références

- [1] JSON Schema <https://json-schema.org>
- [2] JSON Schema Documentation <https://json-schema.org/understanding-json-schema/>
- [3] `json-schema-faker` Générateur (Avril 2023) <https://github.com/json-schema-faker/json-schema-faker>
- [4] `json-everything` Générateur (Avril 2023) <https://github.com/gregsdennis/json-everything>
- [5] `json-data-generator` Générateur <https://github.com/jimblackler/jsongenerator>
- [6] `DataGen` Générateur <https://github.com/wurzy/DataGen>
- [7] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger *Witness Generation for JSON Schema* <https://arxiv.org/pdf/2202.12849.pdf>
- [8] `jschon` Valideur (Avril 2023) <https://jschon.dev/>
- [9] `DataGen` Valideur <https://github.com/networknt/json-schema-validator>
- [10] JSON Schema Test Suite (Février 2023) <https://github.com/json-schema-org/JSON-Schema-Test-Suite>
- [11] Snowplow Dataset (Février 2023) <https://github.com/snowplow/iglu-central>
- [12] WashingtonPost Dataset (Février 2023) <https://github.com/washingtonpost/ans-schema>
- [13] Kubernetes Dataset (Février 2023) <https://github.com/instrumenta/kubernetes-json-schema>
- [14] GitHub Dataset (Février 2023) <https://github.com/sdbs-uni-p/json-schema-corpus>
- [15] Containment Dataset (Octobre 2021) <https://github.com/sdbs-uni-p/json-schema-containment-testsuite>
- [16] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, Stefanie Scherzinger, *A Tool for JSON Schema Witness Generation*, EDBT 2021 : 694-697
- [17] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger, *Not Elimination and Witness Generation for JSON Schema*, CoRR abs/2104.14828 (2021)
- [18] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, Domagoj Vrgoc, *JSON : Data model, Query languages and Schema specification*, PODS'1
- [19] A. Antonio, "Exploitez des données au format JSON", OpenClassrooms. <https://openclassrooms.com/fr/courses/7697016-creez-des-pages-web-dynamiques-avec-javascript/7911021-exploitez-des-donnees-au-format-json>.