Sorbonne Université

Faculté des Sciences et Ingénierie

Master Informatique
Parcours Science et Technologie du Logiciel
(STL)

Benchmarking de solutions optimistes pour génération de données test à partir de JSON Schema

Auteurs

Abdelkader Boumessaoud
Zaky Abdellaoui

Encadrants

Mohamed-Amine Baazizi
Lyes Attouche

Table des matières

1 Contexte et Objectif

JSON Schema [1] est le standard permettant de décrire la structure des données JSON. Son usage se généralise : description des API, vérification de pipelines d'apprentissage, métadonnées en sont des exemples parmi d'autres. D'autres applications, comme l'interopérabilité des microservices, nécessitent la validation de données JSON, sans parler des systèmes de bases de données qui nécessitent de préserver la cohérence des données traitées. Pour pouvoir gérer ces cas d'utilisation, il est crucial de pouvoir générer des instances de tests à partir de schémas donnés en entrée. Or, ce problème est rendu compliqué par l'expressivité du langage, qui combine contraintes structurelles et opérateurs logiques ainsi qu'un mécanisme permettant de définir des schémas récursifs.

Plusieurs implémentations permettent une telle génération automatique, mais ne garantissent pas l'obtention d'un résultat correct, dans le sens où elles peuvent générer des instances non-conformes au schéma en entrée. Elles adoptent une approche optimiste qui consiste à générer une instance en examinant les fragments des schémas et en les combinant dans l'espoir que l'instance obtenue soit valide, et font parfois appel à un validateur externe dans l'optique de "réparer" les instances non conformes.

Le principal objectif de ce projet est de caractériser les limites de ces solutions, lorsque leur code source est disponible. L'idée est d'identifier les classes de schémas JSON traitées correctement et celles qui contiennent des schémas problématiques. Pour atteindre cet objectif, nous avons opté pour une approche de rétro-ingénierie manuelle combinée à une analyse expérimentale, qui nous permet de vérifier au fur et à mesure les hypothèses que nous nous formulons. Le but de l'analyse est également d'identifier les possibilités d'amélioration de ces générateurs et la limite des approches optimistes en tant que telles.

2 JSON Schema

2.1 Description

L'étude et la compréhension de JSON Schema ont été des étapes essentielles de notre projet. Pour ce faire, nous avons consulté la spécification officielle disponible en ligne [2] et consulté un tutoriel agrémenté d'exemples. Nous avons également examiné des exemples de schémas utilisés pour tester les validateurs, afin de mieux comprendre l'utilisation du langage en pratique. Enfin, nous avons eu recours à l'utilisation de validateurs pour vérifier notre compréhension de la sémantique du langage.

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.

```
{
    "type": "object",
    "properties": {
        "firstName": {
            "type": "string",
            "minLength": 3
        },
        "lastName": {
            "type": "string",
            "minLength": 3
        },
        "age": {
            "type": "integer",
            "minimum": 18
        }
    },
     "required": ["age", "firstName", "lastName"]
}
```

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 aplatit et détaillé hiérarchique.

Dans notre projet nous utilisons un validateur se conformant à la spécification et adoptant un "Output format" détaillé et hiérarchique car très utile pour identifier rapidement les erreurs qui sont présentées sous forme d'arborescence et reflète la hiérarchie des propriétés de l'instance.

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"`).

```
{
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        },
        "age": {
            "type": "number"
        },
        "address": {
            "type": "object",
            "properties": {
                "street": {
```

```
                    "type": "string"
                },
                "city": {
                    "type": "string"
                },
                "zipcode": {
                    "type": "number"
                }
            },
            "required": [
                "street",
                "city",
                "zipcode"
            ]
        }
    },
    "required": [
        "name",
        "age",
        "address"
    ]
}
```

```
{
    "name": "John Doe",
    "age": "thirty",
    "address": {
        "street": "123 Main St.",
        "city": "New York",
        "zipcode": "ABCDE"
    }
}
```

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

```
{
    "valid": false,
    "evaluationPath": "",
    "schemaLocation": "https://json-everything.net/e351761edb",
    "instanceLocation": "",
    "details": [
        {
            "valid": true,
            "evaluationPath": "/properties/name",
            "schemaLocation": "https://json-everything.net/e351761edb#/properties/name",
            "instanceLocation": "/name"
        },
        {
            "valid": false,
            "evaluationPath": "/properties/age",
            "schemaLocation": "https://json-everything.net/e351761edb#/properties/age",
            "instanceLocation": "/age",
            "errors": {
                "type": "Value is \"string\" but should be \"number\""
            }
```

```
        },
        {
            "valid": false,
            "evaluationPath": "/properties/address",
            "schemaLocation": "https://json-everything.net/e351761edb#/properties/address",
            "instanceLocation": "/address",
            "details": [
                {
                    "valid": true,
                    "evaluationPath": "/properties/address/properties/street",
                    "schemaLocation":
↪   "https://json-everything.net/e351761edb#/properties/address/properties/street",
                    "instanceLocation": "/address/street"
                },
                {
                    "valid": true,
                    "evaluationPath": "/properties/address/properties/city",
                    "schemaLocation":
↪   "https://json-everything.net/e351761edb#/properties/address/properties/city",
                    "instanceLocation": "/address/city"
                },
                {
                    "valid": false,
                    "evaluationPath": "/properties/address/properties/zipcode",
                    "schemaLocation":
↪   "https://json-everything.net/e351761edb#/properties/address/properties/zipcode",
                    "instanceLocation": "/address/zipcode",
                    "errors": {
                        "type": "Value is \"string\" but should be \"number\""
                    }
                }
            ]
        }
    ]
}
```

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 :

```
{
    "type": "object",
    "properties": {
        "prop": {
            "allOf": [
                {
                    "enum": [
                        "forbidden",
                        "mandatory"
                    ]
                },
                {
                    "not": {
                        "enum": [
                            "forbidden"
                        ]
                    }
                }
            ]
        }
    },
    "required": [
        "prop"
    ]
}
```

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.

`json-everything` Une série de projets [4] visant à étendre la fonctionnalité de traitement des données JSON dans l'espace `System.Text.Json` en C#, qui propose une bibliothèque (elle même en C#) pour interroger et gérer les données JSON. Elle est basée sur un générateur de données pour les classes C# et est donc limitée en termes d'expressivité sur la partie de JSON Schema prise en charge.

`json-data-generator` Une bibliothèque Java [5] pour la génération de données JSON à partir de schémas JSON. Elle prend en charge une variété de types de données JSON, notamment les chaînes de caractères, les nombres, les booléens, les tableaux, les objets et les valeurs nulles. Elle permet également la génération des données aléatoires à partir des schémas JSON, ce qui facilite la création de jeux de données de test.

`json-data-generator` prend en charge la norme JSON Schema Draft 7, mais peut être étendue pour prendre en charge d'autres normes de schéma JSON.

4 Étude des limitations de `json-schema-faker`

4.1 Choix de la librairie

En ce qui concerne `json-data-generator`, cette bibliothèque avait déjà fait l'objet d'une étude préalable documenté [6] qui nous a permis de comprendre le fonctionnement d'un générateur. Nous avons tout de même inclus cette librairie dans les tests réalisés dont les résultats sont présents dans *Tableau 1 : Taux de validité des instances générées (4.4.2)*. Nous observons que le taux de validation des instances générées par `json-schema-faker` et `json-everything` était inférieur à celui de `json-data-generator`. Nous avons décidé d'exclure `json-everything` car il a beaucoup d'erreurs et se comporte de façon très aléatoire rendant quasi impossible la génératlisation des cas où il fonctionne correctement.

4.1.1 Prise en main de générateurs open-source

Nous avons réalisé une étude approfondie du générateur open-source de données JSON `json-schema-faker`. Nous avons pris en main cet outil pour comprendre son fonctionnement et sa syntaxe, puis l'avons utilisé pour générer des données JSON simples et complexes.

Nous avons constaté que ce générateur était capable de générer des données cohérentes et réalistes en respectant les contraintes définies par les schémas JSON. Cependant, nous avons également identifié certaines limites en termes de complexité et de diversité des données générées.

Nous avons notamment remarqué que la génération de données imbriquées ou de structures de données plus complexes pouvait poser des difficultés, et que les options de personnalisation du générateur étaient parfois limitées.

Malgré ces limitations, nous considérons que l'utilisation de générateurs open-source comme `json-schema-faker` peut être une solution pratique et efficace pour la génération de données JSON.

4.2 Choix du validateur JSON Schema

Nous avons examiné plusieurs options disponibles pour les validateurs JSON Schema et avons finalement choisi `jschon` (une bibliothèque Python) [7] pour nos besoins.

Nous avons d'abord étudié la documentation et les exemples fournis par le validateur pour comprendre son utilisation et ses fonctionnalités. Ensuite, nous avons mis en place un environnement de test pour valider nos schémas JSON à l'aide de ce dernier.

Nous avons également exploré ses fonctionnalités avancées, telles que les extensions pour répondre à des besoins spécifiques, pour au final l'intégrer a nos expérimentations.

4.3 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.

4.4 Expérimentations

4.4.1 Chaîne de traitement

Cette chaîne de traitement est une pipeline qui se compose de trois étapes principales.

La première étape est la génération d'instances JSON Schema pour un ensemble de datasets de JSON Schema. Pour ce faire, nous avons utilisé des générateurs open-source tels que (`json-schema-faker` / `json-everything`).

Une fois que nous avons généré les instances, la deuxième étape a consisté à valider ces instances par rapport aux schémas JSON correspondants. Nous avons utilisé un validateur d'instances JSON Schema, l'outil `jschon`.

Enfin, la troisième 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 cette chaîne de traitement, 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.

4.4.2 Collections de schémas utilisés

Lors de notre phase de compréhension des outils de génération de JSON Schema, nous avons décidé d'utiliser le dataset "JSON Schema Test Suite" [8]. Ce dataset contient un ensemble d'objets JSON qui est principalement utilisé par les mainteneurs de bibliothèques de validation JSON Schema pour tester leurs validateurs. Ce qui est avantageux avec ce dataset, c'est qu'il couvre toutes les versions de la spécification JSON Schema, y compris les drafts 2020-12, 2019-09, 07, 06, 04 et 03. De plus, il prend en compte toutes les références JSON Schema telles que les types de données et les opérateurs logiques.

Pour la phase d'expérimentation, nous avons choisi d'utiliser quatre datasets (Snowplow [9], WashingtonPost [10], Kubernetes [11] et GitHub [12]) pour un total de 7573 instances générées. Nous les avons repris d'un article publié. Le choix est ausi justifié par leur intégration dans des domaines applicatifs spécifiques, ce qui les rend plus représentatifs et pertinents pour les tests.

Générateur	Dataset	Total (schémas)	Instance (Valid/Invalid)	Taux de validité (%)
`json-schema-faker`	Snowplow	409	402 / 7	98.28 %
	WashingtonPost	125	102 / 23	81.60 %
	Kubernetes	1082	984 / 98	90.94 %
	GitHub	5957	5254 / 703	88.19 %
`json-everything`	Snowplow	409	307 / 102	75.06 %
	WashingtonPost	125	48 / 77	38.40 %
	Kubernetes	1082	699 / 383	64.60 %
	GitHub	5957	3524 / 2433	59.15 %

Tableau 1 : Taux de validité des instances générées

L'idée principale serait d'utiliser les résultats de la classification effectuée par la pipeline pour travailler sur un grand ensemble de schémas, en considérant des collections de schémas de plus grande taille. Nous pourrions ainsi analyser ceux qui ont des erreurs de validation similaires et utiliser cette information pour valider la performance et la fiabilité des générateurs open-source que nous utilisons, tout en confirmant ou infirmant les hypothèses dérivées sur les collections de plus petite taille.

Dataset	Erreurs	Taux d'occurrence
Snowplow	type	42.86 %
	maxProperties	28.57 %
	minProperties	28.57 %
WashingtonPost	additionalProperties	52.17 %
	type	17.39 %
	required	13.04 %
	enum	13.04 %
	dependencies	4.35 %
Kubernetes	type	82.65 %
	required	16.33 %
	unionType	1.02 %
GitHub	required	34.57 %
	type	17.78 %
	maxItems	7.97 %
	format	6.69 %
	additionalProperties	5.55 %
	...	27.62 %
Total	type	40.17 %
	required	15.99 %
	additionalProperties	14.43 %
	...	29.41 %

Tableau 2 : Taux d'occurrence des erreurs de validation

Cette expérimentation nous permettrait également de mesurer la qualité des schémas et des instances générées par ces générateurs sur un plus grand nombre de schémas, ce qui serait utile pour déterminer leur pertinence dans des projets plus vastes nécessitant une utilisation intensive de schémas JSON et de générateurs de données.

4.5 Limites constatées

Cette partie nous a permis de mieux comprendre le générateur `json-schema-faker` open-source que nous avons utilisé. Nous avons cherché des exemples concrets de schémas JSON pour lesquels ce générateur produit des résultats différents, ce qui nous a permis d'identifier les parties du code qui sont responsables de ces différences.

Nous avons également étudié en détail la documentation du générateur, afin de mieux comprendre son fonctionnement interne et son architecture logicielle. Nous avons ainsi pu identifier les algorithmes et les structures de données clés utilisés par le générateur, ainsi que les méthodes de validation et de génération de données.

Cette approche d'analyse du code guidée par les exemples nous a permis d'obtenir une compréhension approfondie du générateur open-source que nous avons étudié, ainsi que de ses limites et de ses avantages. Elle nous a également permis de formuler des recommandations pour améliorer la qualité des données générées, en identifiant ses parties du code qui pourraient être améliorées ou optimisées.

De plus, une fois une classe de schéma provoquant des erreurs identifiée, nous avons synthétisé des schémas ayant les mêmes caractéristiques (opérateurs, profondeur, etc.) et effectué des tests sur ces schémas pour confirmer l'hypothèse d'une limite de `json-schema-faker` par rapport à une classe de schéma.

4.5.1 Fonctionnement des connecteurs logiques

`allOf` (Conjonction) : Pour valider le `allOf`, les données données doivent être valides par rapport à tous les sous-schémas donnés.

Le traitement du `allOf` dans la fonction `resolveSchema` est équivaut à résoudre des schémas imbriqués dans un objet en fusionnant les propriétés des schémas résolus avec l'objet initial.

Cela permet de créer un objet complet à partir de plusieurs schémas imbriqués.

Cela se fait en vérifiant si une propriété appelée `allOf` est présente dans l'objet. Si `allOf` est un tableau, tous les schémas contenu dans le tableau son résolues en utilisant un appel récursif de la fonction `resolveSchema`. Cette fonction renvoie un nouvel objet qui est fusionné avec l'objet initial en utilisant une fonction `merge`.
Si la propriété `allOf` est toujours présente dans l'objet initial, la fonction récursive `resolveSchema` est appelée pour résoudre les schémas restants.

La fonction `merge` est utilisée pour fusionner deux objets ensemble. Si une propriété de l'objet secondaire existe déjà dans l'objet principal, elle est fusionnée de manière récursive en utilisant à nouveau la fonction `merge`. Si la propriété de l'objet secondaire est un tableau, les valeurs non-dupliquées sont ajoutées à l'objet principal.

`anyOf` / `oneOf` (Disjonction) : `anyOf` et `oneOf` servent à spécifier que plusieurs sous-schémas peuvent être valides pour une instance donnée. La différence entre les deux est que `oneOf` spécifie qu'une seule option doit être valide, tandis que `anyOf` permet que plusieurs options soient valides.

Le traitement du `anyOf` et du `oneOf` dans fonction `resolveSchema` en vérifiant si la propriété `oneOf` ou `anyOf` est définie dans le sous-schéma actuel, et si oui, stocke les sous-schémas dans un tableau. Ensuite, si le sous-schéma actuel contient également une propriété `enum`, le code filtre cette propriété pour ne conserver que les valeurs qui sont valides pour au moins un des sous-schémas stockés dans le tableau.

Ensuite, la fonction retourne un objet qui contient une propriété `thunk`, qui elle-même retourne un nouveau schéma JSON. La fonction `thunk` crée une copie du sous-schéma actuel en retirant les propriétés `anyOf` et `oneOf`. Ensuite, elle choisit aléatoirement l'un des sous-schémas stockés dans le tableau en utilisant une fonction de choix aléatoire. Elle fusionne ensuite la copie du sous-schéma actuel avec le sous-schéma choisi, en utilisant la fonction `merge` détaillée précédemment.

Enfin, elle parcourt les sous-schémas stockés dans le tableau en supprimant toutes les propriétés supplémentaires qui ne sont pas définies dans le sous-schéma choisi. Cela garantit que la copie résultante ne contient que les propriétés nécessaires pour valider l'instance.

`not` (Négation) : Le mot clé `not` déclare qu'une instance valide si elle ne valide pas par rapport au sous-schéma donné.

Le traitement commence par la vérification si le schéma contient la clé `not` et si sa valeur est un objet. Si c'est le cas, il utilise la fonction `notValue` pour créer un nouveau schéma à partir de l'objet en utilisant des valeurs opposées pour les propriétés `minimum`, `maximum`, `minLength` et `maxLength`, et en choisissant un type de donnée différent si le type est spécifié. Ensuite, le code utilise la fonction `clean` qui sert à nettoyer les propriétés non valides du schéma nouvellement créé en utilisant le schéma original. Si le schéma est un objet, la fonction 'traverse' est utilisée pour parcourir les propriétés de l'objet et effectuer la validation.

4.5.2 Recherche et débogage des limites de `json-schema-faker`

Après avoir réalisé une classification élémentaire des erreurs sur nos bases de tests, nous avons décidé de nous pencher d'abord sur la base `GitHub`. Notre démarche fut alors de trouver en s'aidant de la classification des schémas pouvant présenter des exemples de limites logiques de `json-schema-faker`, de formuler des hypothèses sur la source de ces erreurs, puis de faire un travail de recherche en s'aidant du code et d'un débogueur JavaScript, ici celui de l'IDE WebStorm, afin de confirmer ou infirmer nos hypothèses, et éventuellement, si possible, de formuler une éventuelle correction

Calcul du nombre de propriétés : Dans les schémas de test présentant une caractérisation des propriétés par une proposition `oneOf` (OU exclusif), tels que `github/o13131.json`

et `github/o13129.json`, `json-schema-faker` semble incapable de générer des instance respectant les conditions de propriétés requises. Des schémas comprenant une conditionnelle (`if/then/else`), comme par exemple `github/o4834.json` détaillé ci-dessous, produisent également des instances erronées :

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "An example schema",
    "type": "object",
    "properties": {
        "a": {
            "type": "string"
        },
        "b": {
            "type": "integer"
        },
        "c": {
            "type": "string",
            "enum": [
                "abc",
                "def",
                "ghi",
                "jkl"
            ]
        }
    },
    "if": {
        "properties": {
            "c": {
                "enum": [
                    "ghi"
                ]
            }
        }
    },
    "then": {
        "required": [
            "a",
            "b",
            "c"
        ]
    },
    "else": {
        "required": [
            "a",
            "c"
        ]
    }
}
```

Un exemple d'instance que nous obtenons dans ce cas particulier est :

```
{
    "c": "ghi"
}
```

Nous avons alors formulé l'hypothèse que ces erreurs proviennent d'une erreur logique de `json-schema-faker`, qui échoue à combiner une contrainte de type "propriétés requises" avec des

propositions logiques, comme le `oneOf` ou le `if`.

En se penchant sur la source, on trouve que le code coupable se trouve au sein du fichier `object.mjs`. On remarque qu'au lieu de déterminer les propriétés à générer dans l'instance de manière logique, `json-schema-faker` prend une approche empirique en calculant à l'avance un nombre, qui est ajusté au cours de l'exécution et qui possède également une composante aléatoire. En raison de cette approche *ad hoc*, il est difficile de s'assurer de l'exactitude de ces calculs car fondamentalement incomplet, mais nous avons pu trouver des améliorations qui les rendent correct dans plus de cas :

```
// let neededExtras = Math.max(0, allProperties.length - min);
let neededExtras = Math.max(0, allProperties.length - min, min -
↪  requiredProperties.length);

// const _limit = optionalsProbability !== null || requiredProperties.length === max
↪  ? max : random.number(0, max);
const _limit = optionalsProbability !== null || requiredProperties.length === max ?
↪  max : random.number(neededExtras, max);
```

Par exemple, pour `github/4834.json`, on obtient uniquement des instances qui valident contre le schéma, comme :

```
{
    "a": "consequat veniam",
    "b": -77893858,
    "c": "def"
}
```

Cette notion d'approche empirique est importante, puisqu'elle va être la source de la plupart des limitations de `json-schema-faker`.

Conjonction d'une négation avec une affirmative : Dans le schéma `github/o2140`, on remarque qu'un schéma comprenant une négation (mot-clé `not`) accompagnée de la spécification standard d'une propriété génère la plupart du temps des instances erronées. Lorsque l'on réduit le schéma à sa forme la plus simple possible :

```
{
    "type": "object",
    "properties": {
        "prop": {
            "type": "string",
            "not": {
                "type": "number"
            }
        }
    },
    "required": [
        "prop"
    ]
}
```

`json-schema-faker` génère dans la plupart des cas une instance du mauvais type, comme par exemple :

```
{
    "prop": false
}
```

Nous faisons l'hypothèse qu'il existe un dysfonctionnement dans la logique de conjonction implicite entre une négation de contrainte (`"not": {"type": "number"}`) et une autre contrainte (`"type": "string"`). Après débogage, on constate que le problème est en réalité tout autre : l'implémentation de la négation, faite dans `utils.mjs`, l.370-382, est réalisée *ad hoc* au cas par cas, et de manière incomplète. Quand `json-schema-faker` calcule la négation d'une contrainte de type, toute contrainte de type implicitement conjuguée à cette négation est ignorée. Le schéma est donc interprété de la même manière que le schéma ci-dessous, qui renvoie donc un résultat correct :

```
{
  "type": "object",
  "properties": {
    "prop": {
      "not": {
        "type": "number"
      }
    }
  },
  "required": [
    "prop"
  ]
}
```

En se penchant plus longtemps sur le code, on constate un traitement similaire (l.383-391) pour les énumérations (mot-clé `enum`). Nous faisons l'hypothèse que non seulement l'implémentation est également incorrecte pour les énumérations, mais également que l'erreur affecte aussi les conjonctions explicites (mot-clé `allOf`) Pour confirmer notre analyse, nous avons testé le schéma suivant, qui est l'exemple que nous avons utilisée à la section 3 :

```
{
    "type": "object",
    "properties": {
        "prop": {
            "allOf": [
                {
                    "enum": [
                        "forbidden",
                        "mandatory"
                    ]
                },
                {
                    "not": {
                        "enum": [
                            "forbidden"
                        ]
                    }
                }
            ]
        }
    },
    "required": [
        "prop"
```

```
    ]
}
```

Sans surprise, la première proposition est complètement ignorée. `json-schema-faker` semble posséder une implémentation empirique et très incomplète de la négation.

Disjonctions exclusives : Le schéma `github/o81530.json`, qui est un schéma très simple spécifiant un objet et quelques propriétés, produit une instance erronée qui ne vérifie pas une disjonction exclusive (mot-clé `oneOf`). Afin de formuler une hypothèse, nous avons alors simplifié au maximum le schéma tout en gardant la même structure et la même erreur :

```
{
    "type": "integer",
    "oneOf": [
        {
            "maximum": 0
        },
        {
            "maximum": 200
        }
    ]
}
```

Avec ce schéma en entrée, `json-schema-faker` ne produit que des entiers négatifs, qui satisfont les deux contraintes exclusivement disjointes, et ne sont donc pas valides. Nous faisons donc l'hypothèse que la logique du OU exclusif (`oneOf`) est problématique. En se penchant sur le code, on découvre que cette implémentation logique n'existe tout simplement pas (`buildResolveSchema.mjs`, l.103) : dans le cas général, les `oneOf` sont traités de la même manière que les disjonctions non exclusives (`anyOf`), en choisissant simplement une contrainte au hasard, ce qui conduit `json-schema-faker` à produire des instances fausses dans la quasi totalité des cas. Le seul cas de disjonction exclusive géré est implémenté comme un cas particulier, en la présence d'une énumération (l.108), qui sont traitées par une méthode de `utils.mjs`, qui s'assure que l'instance valide bien une seule contrainte du `oneOf`.

Propriétés requises mais non explicitées : En analysant les résultats des bases de tests, nous avons également pu constater une limitation plus mineure, mais néanmoins source de nombreuses erreurs, liée au respect des propriétés requises par le schéma. Pour formuler une hypothèse, nous avons simplifié ces occurrences en un schéma comme celui-ci :

```
{
    "properties": {
        "spec_req1": {
            "type": "string"
        },
        "spec_req2": {
            "type": "string"
        },
        "spec_notreq1": {
            "type": "string"
        },
        "spec_notreq2": {
            "type": "string"
        }
    },
    "required": [
        "spec_req1",
        "spec_req2",
        "notspec_req1",
        "notspec_req2",
```

```
        "notspec_req3"
    ],
    "type": "object"
}
```

`json-schema-faker` génère ici dans la plupart des cas des instances erronées, qui ne contiennent pas les cinq propriétés requises par le schéma, comme la suivante :

```
{
    "spec_req1": "in cillum magna id",
    "spec_req2": "eiusmod aliqua",
    "spec_notreq1": "exercitation est dolore",
    "spec_notreq2": "in adipisicing sed",
    "notspec_req1": -1009337.5116580427
}
```

Il semble que `json-schema-faker` soit incapable générer une propriété `required` si elle n'est pas explicitée dans le champ `properties`. Après vérification des sources et débogage, on découvre qu'il s'agit à nouveau d'une conséquence de l'approche empirique choisie par les développeurs. Dans `object.mjs` (l.219), le code s'arrête prématurément d'ajouter les propriétés non explicitées, en raison du calcul incomplet du nombre de propriétés à générer.

4.6 Remarques

Notre travail sur les limitations de `json-schema-faker` et notre analyse de son code nous a permis d'avoir une bonne idée des paradigmes utilisés lors de sa conception. Au lieu de prendre une approche logique, en cherchant à implémenter les opérateurs (`oneOf`, `allOf`, `not`, etc.) eux-mêmes, et de générer des instances en faisant des calculs de propositions, `json-schema-faker` est beaucoup plus empirique.

En effet, le code de la génération est implémenté de manière beaucoup plus empirique qu'attendu, avec des disjonctions de cas et énormément de conditionnelles. Ce dernier point a pu être un obstacle quant à la lisibilité du code, puisqu'une telle approche nécessite la gestion d'une grande quantité de cas et de branche pour atteindre une performance (en termes de validité des instances générée) satisfaisante.

De plus, cette approche explique en grande partie l'incomplétude de l'implémentation de `json-schema-faker`. Des fonctionnalités très simples de la spécification de JSON Schema sont manquantes ou mal gérée, non pas à cause d'erreurs, mais à cause d'oublis ou de cas manquants parmi ceux qui doivent être traités pour atteindre un résultat satisfaisant.

En outre, nous avons jugé `json-schema-faker` manquait de documentation et de commentaires clairs, utiles à la compréhension, la maintenance, et la rétro-ingénierie de l'outil.

5 Perspectives

L'une des contraintes principales de notre projet a été le manque de temps. En effet, nous aurions aimé approfondir plusieurs aspects de notre travail.

Tout d'abord, nous avons dû investir beaucoup de temps sur les maîtriser et la compréhension du langage JSON-Schema et des outils de validation et de génération leur fonctionnement du fait que, ces technologies étant nouvelles pour nous

Nous aurions souhaité aller plus en profondeur dans l'analyse des erreurs en expérimentant sur un plus grand nombre de schémas et d'instances pour mieux comprendre les erreurs les plus fréquentes et éventuellement trouver des solutions pour y remédier.

Enfin, nous aurions aimé analyser et expérimenter d'autres générateurs open source afin de pouvoir faire une étude comparative approfondie de différentes solutions de génération de données aléatoires.

6 Conclusion

Après avoir mené une étude approfondie des générateurs open-source, nous avons pu caractériser les limitations de l'un de ses générateurs en termes de classes de schémas JSON traitées correctement et de classes de schémas problématiques. Nous avons utilisé des méthodes de rétro-ingénierie et des analyses expérimentales pour évaluer l'efficacité et les limites.

L'analyse des résultats nous a permis de caractériser les classes de schémas qui posent des problèmes aux générateurs opens source, ainsi que d'identifier les domaines dans lesquels ce générateur peut être amélioré. Nous avons constaté que les générateurs open sources actuels peuvent traiter efficacement de nombreux types de schémas JSON, mais qu'ils peuvent rencontrer des difficultés avec certains types de schémas plus complexes.

En fin de compte, nous espérons que notre projet pourra fournir des informations intéressantes pour les développeurs de générateurs open-source et les utilisateurs de ces outils, en mettant en évidence les limites actuelles des générateurs et les domaines dans lesquels des améliorations sont nécessaires. Nous espérons que ces résultats contribueront à améliorer la qualité et la fonctionnalité des générateurs open-source, ce qui bénéficiera à la communauté des développeurs et des utilisateurs de logiciels.

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] 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

[7] `jschon` Validateur (Avril 2023) https://jschon.dev/

[8] JSON Schema Test Suite (Février 2023) https://github.com/json-schema-org/JSON-Schema-Test-Suite

[9] Snowplow Dataset (Février 2023) https://github.com/snowplow/iglu-central

[10] WashingtonPost Dataset (Février 2023) https://github.com/washingtonpost/ans-schema

[11] Kubernetes Dataset (Février 2023) https://github.com/instrumenta/kubernetes-json-schema

[12] GitHub Dataset (Février 2023) https://github.com/sdbs-uni-p/json-schema-corpus

[13] 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

[14] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger, *Not Elimination and Witness Generation for JSON Schema*, CoRR abs/2104.14828 (2021)

[15] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, Domagoj Vrgoc, *JSON : Data model, Query languages and Schema specification*, PODS1

[16] 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.