Overview and Perspectives for Optimistic JSON Schema Witness Generation

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

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL Research University
dario.colazzo@dauphine.fr

ABSTRACT

JSON Schema is an expressive schema language for describing JSON documents which combines structural and Boolean operators, and features negation and recursion. One of the main problem related to JSON Schema management is that of checking whether a schema admits at least one instance, and this can be solved by approaches aiming at generating witnesses for the schema. In this context, *optimistic* witness generation is characterized by the fact that witnesses are first generated for each fragment of the schema hoping that their trivial combination, according to schema indications, form a global witness. If this is not the case, then corrections may be made.

Optimistic witness generation is attractive in that it is likely to be fast, even if ensuring completeness is impossible without incurring exponential time blowup [**?**]. Several implementations are available but only few are open-source and, more interestingly, none has a documentation that facilitates studying its capabilities and limitations. To overcome this shortcoming, we undertook the daunting task of reverse-engineering the most relevant tools in the domain. We assessed the soundness and completeness of these tools and report on our experience in analyzing important features by revealing strong points, limitations and possible improvements, so as to pave the way to the design and implementation of more powerful generation techniques.

KEYWORDS

JSON, JSON Schema, Data Generation

1 INTRODUCTION

JSON Schema is a language for describing families of JSON documents. It is increasingly adopted simply because the number of APIs using JSON as a format for exchanging data explodes. In this context, it is paramount to be able to decide whether a given schema is *satisfiable* and to enable for generating testing data starting from a JSON Schema specification. This ability may serve many applications: for instance, one can simulate the behavior of a program on synthetic data in order to anticipate potential mismatches or performance degradation whose cost may have dramatic consequences on the reliability of services.

We have recently worked on approaches to generate a witness for a JSON schema, under the condition that *uniqueItems* for array description is not used [**?**]. The approach is sound and complete, but currently is not able to generate a number of witnesses (instances) given by the user, and for schema featuring particular operators, generation time could be prohibitive (witness generation has been proved to be EXPTIME complete for JSON Schema minus *uniqueItems* [**?**]). This is due to the fact that our approach rewrites the schema, before generation, into a novel form of disjunctive normal form that on one side is time expensive to generate, and on the other hand is expensive to explore. One alternative approach that we plan to follow, and that we dub *optimistic* example generation, is based on avoiding *expensive* schema rewriting before generation, and on generating witnesses (examples) for schema fragments and then opportunely adapt witnesses so that the global schema is satisfied. The idea is to have an approach that is faster, even for a part of schemas featuring operators complex to deal with, by scarifying completeness, but by having at the same time the possibility to relay on an approach that best fits the problem of generating not one witness/example, but many of them. Fastness is particularly important in this context.

There are already implementation of systems of this kind that are publicly available. So a first important step is to understand their design principles and experimental behaviour, so as to understand their strengths and limitations, and pave the way for the design and implementation of more effective systems. The study we present here offer a comparative analysis of these systems, and discuss design principles of alternative, more effective systems for JSON Schema example generation.

*** the part below needs to be stabilized***

***** We discuss three systems [], that after a preliminary analysis have revealed to be to most impactful ones. More precisely, we assess, for each solution, it completeness in two settings: in the absolute case where we consider hand-written schemas and in real-life by using five collections of real-world schemas. So in this paper we make a first step towards our future goals, that include i) improving the technique underlying [**?**] in order to obtain an approach with a wider scope, and ii) identify static analysis techniques allowing for statically deciding whether for a given schema it is better to use the new, incomplete, but faster approach rather than our [**?**] complete, but potentially slower approach. This new approach would entail improvements in terms of execution time wrt [**?**], while still being complete.

amine: we need to include UniqueItems in the comparison

The paper is structured as follows. After having given a brief introduction to JSON Schema in Section 2, we present and discuss solutions, advantages and limitations of optimistic witness generation in Section 3, by focusing on [**?**]. In Section 4 we present some insights obtained by means of an experimental evaluation, before providing concluding remarks.

amine: adapted from vldbj

2 OVERVIEW OF JSON SCHEMA

Many versions of JSON Schema have been released, the latest dating to December 2020, but the most widely adopted version is anterior to that. A JSON Schema specification is itself a JSON document describing assertions of several kinds:

- Base values assertions are meant to describe strings or numbers: a regular expressions ("`pattern`") can be used for describing valid strings, while an interval ("`minimum`", "`maximum`",...) and a set of multiples ("`multipleOf`") are a convenient way to capture the set of allowed numbers.
- Array assertions are used for specifying the types of the elements that can appear in an array, for restricting the minimum and maximum size of the array ("`minItems`", "`maxItems`"), for bounding the number of elements that satisfy a given property ("`contains`", "`minContains`", ...), and for enforce uniqueness of the items ("`uniqueItems`").
- Object assertions are used for specifying the schema of properties designated by their names or by regular expressions capturing their names ("`properties`", "`patternProperties`"), for requiring the presence of some properties ("`required`"), for limiting the number of properties ("`minimum`", "`maximum`"), and for express complex dependencies among properties ("`dependencies`").

Besides these capabilities, the language allows for building complex expressions using Boolean operators like disjunction ("`anyOf`"), conjunction ("`allOf`"), and even negation ("`not`"). The language has also a variable definition mechanism using the "`definitions`" and "`$ref`" keywords which makes it convenient to reuse schema fragments within the same or an external schema. As a side effect, it is possible to specify a schema with recursive variables.

Example 1. *To give an overview of assertions that can be expressed in this language, consider the following schema which combines, using a disjunction, a sub-schema capturing string values starting with 'a', and a sub-schema accepting arrays, which are potentially empty, and if not empty, must contains numbers multiple of 7.*

```
{ "anyOf":[
  {"type": "string", "pattern": "^a.*" },
  {"type": "array", "contains": {"type":"number",
                                 "multipleOf":7}}] }
```

lyes: (optional rewording): should we mention that the schema can either accept an empty array or any array whose items are numbers multiple of 7

amine: nice remark. I changed the example and the comment

3 JSON DATA GENERATION USING JSONGENERATOR

jsongenerator is an open source Java implementation available on GitHub [**?**] and bundled as an online demonstrator [**?**]. Since no formal description of this tool was available, we performed our study by both inspecting the source code and running the code on specific examples to test some hypotheses.

The tool adopts a try-and-fail approach: first, an instance is generated from the input schema, then it is validated against the schema using an external *validator*. In case of validation failure, the instance is fixed by exploiting the validation error messages and submitted again to validation. The fixing-validation process may loop until reaching a state where the instance is deemed valid. A simple safety condition is introduced to avoid infinite loops: when fixing fails at dealing with the same error twice, a failure is raised and the process is aborted.

dario: should add here that we run it on thousands on schemas too...?

The generation and the fixing mechanisms are complimentary to each other: while generation is optimistic at generating an instance starting from the schema, fixing is meant to address the potential problems that generation left unsolved and relies on the validation errors to apply local modifications so that the instance becomes valid to the original schema. We explain the logic underlying each mechanism then analyze its limitations through examples that were used to validate our assumptions.

amine: I moved the limitations of the tool to a dedicated section

3.1 The generation mechanism

As already mentioned, the generation mechanism adopts an optimistic strategy which consists in making the best guess out of the assertions present in the schema. The underlying mechanism is rather straightforward and is guided by the structure of the schema by associating to each kind of typed-assertions, i.e. those describing numbers, strings, objects and arrays, a specific generation mechanism that adheres to the logic of this assertion. The case of Boolean expressions is very naive and relies on choosing a random branch of binary operators (`allOf`, `anyOf` and `oneOf`) and totally ignoring the `not` operator. Therefore, the generation almost always fails when conjunction or negation are involved and the fixing mechanism is not always able to compensate this limitation as we will show.

Now, we comment on the generation logic adopted for each family of assertions.

3.1.1 Base values assertions. The generation for strings assertion does not present any challenge and relies on a built-in *Javascript* function that generates a string of arbitrary length starting from a regular expression. The generation for numbers is also straightforward and needs to care about the interval of the "`multipleOf`" constraint and the bounds given by `minimum` and `maximum`. The generated value is obtained by the following formula $value = x \ \% \ (maximum - minimum) + minimum$ where x is a random double value in the range $[0, 1]$. The treatment of integers refines the one for numbers but is not described here.

amine: is there any flaw in the implem of this category? If so, report.

3.1.2 Object and array assertions. The generation for objects and arrays is technically more involved because object and array assertions are usually built from arbitrary sub-schemas that can be complex in turn, and because of possible interactions between sub-schemas present in the same assertion.

Example 2. *To illustrate the potential problem that can arise while dealing with object assertions, consider the following schema which describes an object whose compulsory property "a" must be associated to a string satisfying two patterns* `a(c|e)` *and* `a(b|c)a`. *The generation mechanism fails at detecting this interaction and generates an invalid object because the property "a" is associated to a string that only satisfies one pattern but not both.*

```
{ "type": "object",
  "properties": {
    "a": {"type": "string", "pattern": "a(c|e)"}},
  "required": ["a"],
  "patternProperties": {"^a$": {"type": "string",
        "pattern": "a(b|c)a" } }}
```

The general management of objects is rather simple and relies on determining the length of the object to generate by examining the `minProperties` operator. For objects of length $l > 0$, the assertions "`properties`" and `patternProperties` are examined in order to collect a set of (*key*, *schema*) candidates such that when *key* appears as a label of "`properties`" or is satisfied by a pattern of `patternProperties`, than its associated *schema* satisfies the constraints of the corresponding assertions. The generation is recursively applied to each *schema* of the pairs to produce an object satisfying the complex assertion.

The approach is not able to deal with the situation illustrated in Example 2 simply because it is not able to combine arbitrary expressions, a problem that is consistent with the limitation to deal with conjunction.

Arrays can be seen as objects where properties have an index instead of a label. However, arrays are slightly more complex than that because of possible interactions with `contains` and `uniqueItems` whose presence may introduce extra complexity. To generate an array, first its length l is determined by examining `minItems` and `maxItems` if they exist, otherwise this length is set by reading specific parameters or will simply be set to 0, meaning that an empty array will be generated. In the general case ($l > 0$), the sub-schemas of the array assertion (those appearing in `items` and `contains`) are collected in a list of potential generators for elements of the array. This list is filtered to get rid of non satisfiable schemas using a simple syntactic verification and potentially enriched with the sub-schema of `additionalItems` in case the schemas of the candidate set are not sufficient to generate l items. In a subsequent step, generation is guided by this list and consists in recursively generating a value from each schema in the list while checking for the existence of already generated values, in case `uniqueItems` is involved. This approach poses a major limitation as the length of the generated array is determined by the size of the list of collected schemas while the upper-bound `maxItems` is completely ignored. We confirmed this limitation by testing the tool on a schema combining `items`, `contains` and `maxItems` as illustrated with the following example,

Example 3. *The following schema admits arrays that have exactly one element of type string which satisfies the pattern "a(b|c)a". Despite this fact, the generated array contains two strings (an arbitrary string and a word of the language "a(b|c)a") and totally ignores the* `maxItems` *limit. As we will discuss later, the fixing phase can not remediate to this case.*

```
{ "type": "array",
  "items": [ {"type": "string" } ],
  "contains": {"pattern": "a(b|c)a",  "type": "string" },
  "minItems": 1,  "maxItems":1
}
```

3.1.3 Other capabilities. The generation mechanism can be parameterized by setting several parameters such as the size of the output instance, the possibility of generating extra properties for objects that are not limited to the required ones nor to those that are constrained by the `properties` and `patternProperties` operators. While these parameters can turn useful by extending the capabilities of the tool to act as an example-generation tool, they do not improve the soundness of the approach adopted by the tool which, again, suffers from strong limitation when it comes to dealing with conjunction and negations.

3.2 The fixing mechanism

The fixer is called when generation returns a document that partially matches the schema and it exploits the validation errors to repair the instance. This reparation is possible because most errors are explicit enough to suggest a direct resolution of the error but also because the used validator is exhaustive and returns all errors that are detected during validation. To each error, the validator indicates the assertion that is not valid but also the location of the violating values in the generated document. This makes the process of fixing very convenient, since it suffices to associate a resolution rule to each kind of error such that each violating value is substituted with a value that is expected to satisfy the assertion involved in the error. As explained earlier, many fixing-validation cycles may be required to completely fix the instance.

We discuss the resolution adopted for each kind of expression.

3.2.1 Resolution of base value errors. The resolution rules for base values are fairly simple and rely, in most of the cases, on the same mechanism for generating base values from assertions. We only report on a set of interesting rules that slightly differ from the case of generation. For instance, resolving a *multipleOf* error relies on the same logic used for generation with the difference that the seed value is not chosen randomly but is set to the violating value so to avoid regenerating a similar non-conforming value. Resolving string errors related to patterns and format amount to generating a string starting from a regular expression.

3.2.2 Resolution for complex assertions. Validation errors for objects can be related either to missing properties or dependencies, or violation of upper bounds. To fix a missing property or a dependency error, it suffices to add a pair *k:0* where *k* is the name of the missing property. If we generated an object with less properties than required (*minProperties* error), first we retrieve the value of *minProperties*, and add to the current *object* the pairs *p:0* with *p* a property appearing in the *properties* block of the schema (i.e. if *p* is already in the current *object*, only its value may change). Afterwards, if we still have less properties than required, if the schema does not contain a *patternProperties* block, we simply generate random keys and add them to the current *object* with each one of them having the value *0*, otherwise we generate keys that satisfy patterns of the *patternProperties* block with values satisfying the corresponding schemas. To satisfy the upper-bound *maxProperties* constrain the fixer simply removes a set of properties from the current *object*.

For arrays, errors may be related the *uniqueItems* assertion, to arrays not reaching the minimum or having more elements than needed, or to elements not conforming to the type indicated in the `items` assertion. To satisfy the *uniqueItems* assertion, the strategy is to generate a fresh array hoping that it does not contain duplicates. To reach the minimum number of items, a non-valid array is extended at the head with a default value until reaching the required lower-bound. However, no specific resolution is dedicated to dealing with the upper-bound being exceeded since removing of elements is not considered nor the possibility updating existing elements.

lyes: didn't know how to correct the typo: nut ??

dario: should we here make a reference to our object example, and precise whys this fixing approach does not work for that example?

amine: yes but the fixing does not produce a valid instance :(. We need a working example

3.2.3 General expressions. The resolution of violations due to the logical operators `anyOf`, `oneOf`, are fairly simple and consists in randomly selecting one branch to fix. Violations due to negation are not considered and deal with by returning a default value (the number 0). Violations related to a *type* error are dealt with by randomly choosing a type among the suggested ones and based on the selected type, return its default value: (an empty array/object/string, null, false or the 0).

dario: here we should be carefoul, negation is hidden inside oneOf

3.2.4 Limitations . The validation-fixing approach suffers from many limitations which are related to a lack of a support for specific errors like those due to negation and to the `maxItems` bound in arrays. We believe that relying on fixing to deal with negation is not a viable solution since it may require to test a very large set of solutions without even concluding. In contrast, dealing with the `maxItems` limitation is feasible and needs to adapt the current solution to decrease the elements in an array. Another serious limitation is the resolution strategy itself which is non-deterministic in the order of dealing with validation errors but also because it adopts a too simple fix-point condition that does not even prevent infinite loops like we illustrate with the following example.

Example 4. *Consider the following schema which captures arrays containing at least one number such that it is equal or above 2. The generation, which does not support conjunction, produces an array with a random number below 2 and triggers a validation error pointing to the* `allOf` *assertion. To fix this error, the fixer adds the number 2 to the head of the array while keeping the original non-valid number in the tail. This fixing strategy leads to an infinite loop increasing the array infinitely without removing the non-valid element.*

```
{ "type": "array",
  "items": {"allOf": [{"type": "number"},{"minimum": 2}]},
   "minItems": 1}
```

In our presentation we deliberately focused on uncovering the limitations of the tool since, our aim, is to make an informed decision about the utility of the tool in the general case. We balance our point of view with a more objective study based on an experimental analysis of the tool correctness using real-life schemas.

4 EXPERIMENTS

4.1 Goals

The goal of the experiments is twofold. We first show that *Data generator (*DG*)* outperforms, in general, its competitors in terms of soundness of the generated witnesses. Second, we report on its theoretical limitations, experimentally. We reuse the schema collections of [**?**] since it has been curated and is diverse and large enough to allow for generalizing on the exact limitations of any tool. The experiments were carried out on a Latitude e6410 laptop with a 4-core Intel i5 2.67GHz CPU, 8 GB of RAM, running Linux Mint 19.1, using version 0.4.6 of *Data generator (*DG*)* instrumented so that to log the two phases of the generation process and, more importantly, to return a fine-grained account of the fixing mechanism, when it takes place. We fixed the number of attempts to 1,000 to avoid into infinite time.

4.2 Schema collections

PUT

amine: description by indicating that it was used in a prior work

Collection	#Total	#Sat/#Unsat	Size (KB): Avg/Max
GitHub	6,427	6,387/40	8.7/1,145
Kubernetes	1,092	1,087/5	24.0/1,310.7
Snowplow	420	420/0	3.8/54.8
WashingtonPost	125	125/0	21.1/141.7
Handwritten	235	197/38	0.1/109.4
Containment-draft4	1,331	450/881	0.5/2.9

Table 1: Description of the schema collections

4.3 Competitors

Despite the large number of tools which pretend generating data from JSON Schema, only few are worth being considered. Specifically, we restrict to recent open-source tools that are still active projects on Github and found only two implementations that fulfill our requirements.

4.3.1 JSON Faker (*JF*) *[**?**]* . It is Javascript implementation of a rather sophisticated, yet, incomplete, generator that is also bundled as an online tool and is being used by the community, as witnessed by the ongoing activity on GitHub. For instance, many issues are submitted to request or suggest supporting new features like negation. The tools adopts a straightforward approach which behaves rather well with structural types but, as expected, show limitations when it comes to dealing with conjunction and negation, in the general case.

4.3.2 JSON everything (*JE*) *[**?**]* is part of a library for querying and managing JSON data written in C#. It is based on a data generator for C# classes and is thus, limited in terms of the expressivity of the fragment of JSON Schema that is supported because object-oriented design does not support union type nor negation .

amine: to be confirmed - more explanation. Report on limitations stated in the documentation

4.4 Comparative analysis

We experimented the three tools on the six collections. For each schema, we generated an instance and verified its validity against the original schema using an external validator [**?**]. We measured the correctness ratio for each combination of tool and schema collection and report the results in Table 2. More precisely, we indicate the ratio for schemas that are processed and yield a valid witness (*Success*), that of schemas that are processed by yield a non-valid witness (*Error*), and those that encounter a runtime error during processing (*Failure*).

We notice that *DG* and *JF* achieve far better results than *JE*, which was rather expected, and that *DG* dominates *JF* on four of the six collections while for two collections (Snowplow and Handwritten), *JF* takes slightly over *DG*. To understand the reason of this non uniformity, we examined the schemas of Snowplow and Handwritten where *DG* fails while *JF* succeeds and realized that ....

amine: still to be completed

Collection	Tool	Success	Failure	Logical Errors
GitHub	*DG*	**94.21%**	2.86%	2.93%
	JF	82.64%	5.80%	11.56%
	JE	58.68%	20.77%	20.55%
Kubernetes	*DG*	**99.54%**	0%	0.46%
	JF	89.84%	0.18%	9.98%
	JE	69.41%	8.43%	22.16%
Snowplow	*DG*	94.76%	0%	5.24%
	JF	**95.24%**	2.62%	2.14%
	JE	72.38%	18.10%	9.52%
WashingtonPost	*DG*	**96.80%**	0%	3.20%
	JF	87.20%	0%	12.80%
	JE	29.60%	25.60%	44.80%
Handwritten	*DG*	8.51%	34.04%	57.45%
	JF	**9.36%**	17.45%	73.19%
	JE	3.83%	1.70%	94.47%
Containment-draft4	*DG*	**28.77%**	30.88%	40.35%
	JF	27.20%	2.78%	70.02%
	JE	0.23%	3.91%	95.86%

Table 2: Comparison of the correctness results.

4.5 Analysis of *DG*

To better assess the limitations of *DG*, we analyzed the performance of its two phases separately. We first assessed the ability of its *generation* module to produce a correct instance w/o resorting to fixing and analyzed the ability of the *fixing* module to compensate for the limitations of the generation. To do so, we report, in Table 3, for the schemas that pass, those that pass during the generation (first column) or require fixing (second column), and among the schemas that fail, as per the external validator [**?**], those that are not detected to be non valid by the *DG* validator and are, thus, not subject to fixing (third column), and those for which fixing does not lead to synthesizing a valid witness (last column). Interestingly, for real-world schemas where the *DG* returns good results, in more than 86.40% of the cases, generation is enough to generate a correct instance while the rate of fixing in this category of schemas ranges from 0% to 10.40%.

amine: does combining the generation of both tools with the fixing improve correctness?

Collection	Success		Logical Errors	
	no-fixing	fixing	no-fixing	fixing
GitHub	91.58%	2.63%	0.34%	2.59%
Kubernetes	99.54%	0%	0%	0.46%
Snowplow	93.10%	1.66%	0%	5.24%
WashingtonPost	86.40%	10.40%	3.20%	0%
Handwritten	4.26%	4.26%	3.83%	53.61%
Containment-draft4	1.35%	27.42%	0%	40.35%

Table 3: Analysis of the two phases of *DG*

Our last study aims at explaining the reasons preventing the generation and the fixing phase from succeeding. We analyzed logs collected during the execution of *DG* which we instrumented in a way to allow for tracking the interaction between the generation the fixing and the internal validator. This log reports on the potential validation errors and on the series of the attempts to fix the instance while exploiting these errors. We found the most significant error to be related to oneOf with overlapping branches. In such a situation, the *DG* which choses a branch randomly, is likely to generate an instance that satisfies both branches. Fixing can not useful in this situation because the old instance can not be removed.

amine: ongoing analysis

amine: no longer sure about this fact

5 CONCLUSION

In this paper we studied the witness generation problem under a purely practical perspective by analyzing an existing open-source solution that was developed without an in-detph theoretical background. Our study allowed us to reveal the major limitations of this approach and, more importantly, to confirm that good implementation principles are not sufficient to address a fundamental problem whose complexity has been shown to be exponential. Nevertheless, this study allowed us to consider the problem of witness generation under a new perspective in which, simple optimistic solutions can be used to deal with the majority of cases that do not present any complication while reserving the more sophisticated approach we defined in [**?**] to situations requiring a thorough analysis of the schema. This study led us to discover concrete limitations of optimistic solutions and to consider the study of means to characterizing situations that require the use of a principled approach of witness generation. A possible idea is to adopt a static analysis approach to classify schemas based on their willingness to be correctly processed by an optimistic tool. Such an analysis would benefit from the limitations we already discovered like the use of specific assertions or the co-occurrence of specific operators. Of course, a static analysis is not meant to be exact but to provide a good over-approximation of the decision to adopt an optimistic solution or not ; a trade-off between the complexity of this analysis and the gain in terms of running time needs to be considered.

A PERSONAL COMMENTS/QUESTIONS

- a descent witness generation approach should be able to merge assertions, to push negation, and to combine constraints and requests. We should always investigate these capabilities when studying external tools.
- Does validation help in compensating for limitations in and-merging, negation pushing and constraint-request combination?
- A more fundamental question: is repairing an instance against a schema always possible?
- Is a lazy approach deprived from am, np, cr-comb which generates an instance than repairs based the instance feasible? (an approach similar to that of datagen)
- potential related work: database repair, logical program fixing
- Strategic question: survey or journal-style is probably bette suited for this kind of study, if extensive.
- may be we should restate the motivation as being a study of practical tools for witness generation that have proven rather satisfactory on real world schemas despite featuring some limitations. analyse the schemas that do not pass in the tools (intersection, union, ...). tell whether combining two open-source tools can achieve completeness
- present results on RW and hand-written HW as complementary: while HW helps figure out limitation, RW helps understand the extent of this limitation and its practical impact

B REVIEWS

B.1 Reviewer 1

- SCORE: -1 (weak reject)
- R1.1. The study is performed by inspecting the source code, and some considerations are reported in this section. It is not very clear at the end of this section to understand exhaustively which mechanisms are taken into account, and which are not?
- R1.2. The experiments are very short. The datasets used are two in number, with no real explanation of the choice of these two datasets. The experimental protocol is not clearly and in detail presented.

B.2 Reviewer 2

- SCORE: 1 (weak accept)
- R2.1. The topic of this paper clearly fits with BDA's approach. It is relatively interesting, though I really have to take the author's word about how useful the problem is (I would rather think that, while satisfiability makes sense for logical formulas, knowing whether a schema is satisfiable hardly bring relevant information because one would generally expect a schema to satisfy much stronger properties before it is considered useful). On the other hand, being able to generate a witness may perhaps be also/more useful as a way to help "visualize" the schema on a concrete instance.
- R2.2. investigate more solutions
  - https://www.liquid-technologies.com/online-schema-to-json-converter
  - jsonbuddy
  - fake-schema-cli

B.3 Reviewer 3

- SCORE: 0 (borderline paper)
- R3.1. Une section 4 plus détaillée serait très intéressante en donnant d'avantage d'information sur le corpus (au lieu d'un renvoi à un autre article) et surtout plus de détails sur les cas problématiques pour cette approche directe et leur utilisation dans des schémas "réellement utilisés".