CONTENTS

A Tool for 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

Francesco Falleni
Dipartimento di Informatica, Università di Pisa
fallenifrancesco98@gmail.com

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa
ghelli@di.unipi.it

Cristiano Landi
Dipartimento di Informatica, Università di Pisa
c.landi7@studenti.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

Stefanie Scherzinger
Universität Passau
stefanie.scherzinger@uni-passau.de

ABSTRACT

JSON Schema is an evolving standard for the description of JSON documents. It is an extremely powerful language endowed with boolean operators and recursive definitions. Hence, classical problems like schema *consistency* and *equivalence* may be challenging without well-principled tools. Based on our recent effort for laying down an algebraic formal semantics of JSON Schema, we demonstrate an approach for generating valid *witnesses* of a user-defined schema. Our goal is not only to allow programmers to design schemas that meet their intentions, but also to guide them in their journey to understanding the semantics of existing schemas, in an interactive fashion. We thus aim to contribute to the adoption of the JSON Schema language by facilitating its use.

1 INTRODUCTION

In recent years, JSON has become the *de facto* standard data interchange format, and is now widely used for exchanging data between web applications and remote servers, for exporting and importing data, as well as inside complex ML pipelines for combining different stages, as in Google TFX [11].

Despite its great popularity, there is no consensus about a *standard* schema language for JSON yet. Indeed, in many cases, JSON datasets come without a schema, and the end user or application has the duty to infer or guess a new schema, if required. In many other cases, however, several and vastly different schema languages are used for describing the structure of JSON data, ranging from Apache Avro [3], to the MongoDB internal schema language [6], and to JSON Schema [12].

Differently from what happened with XML, whose standard schema languages (DTDs and XML Schema) reached quickly a wide diffusion, JSON Schema is not being adopted at the same pace. Many reasons are slowing down its adoption, but, according to our observations, a major obstacle is the fact that, while extremely powerful, JSON Schema is – frankly – hard to use. Indeed, a schema is a logical combination of implicative assertions, and some of them may produce side effects on previous ones.

As a consequence, leaving the realm of plain vanilla schemas may expose the programmer to many risks, such as the definition of a schema with unintended semantics, or one that is even empty.

Example 1.1. Consider the following schema.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "required": [ "x" ]
}
```

This schema declares that all instances are JSON objects, and that each object has a mandatory *member* whose name is x and whose type is *integer*. This schema, however, does not impose further constraints on object values; therefore, an object may also have supplementary and unconstrained members.

Example 1.2. Consider now the following schema, differing only in the next-to-last line.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "not": { "required": [ "x" ] }
}
```

One may assume that this specifies that x is "not required", hence is optional. However, given the semantics of JSON Schema, negating a required member does not make it optional: indeed, the final effect is to actually forbid the presence of the member, hence excluding any JSON object having a member whose name is x (this example is inspired by a discussion on Stack Overflow [1], where the confusing effect of this schema is testified).

Given the complex and non-trivial interplay between schema assertions, designing a rich yet sound schema is challenging, especially when other powerful mechanisms of JSON Schema are involved, such as negation, mutual exclusion, recursion, union and conjunction, as well as array constraints controlling array length and content, possibly requiring uniqueness of elements.

Motivating Witness Generation. The state-of-the-art approach for exploring JSON Schema semantics is ultimately a manual trial and error: using a JSON Schema validator, a schema designer can test whether a JSON document is valid w.r.t. the schema. That is, the designer must come up with suitable *witness* documents.

Yet, in this demo, we present a tool capable of automatic witness generation. For instance, for the schema from Example 1.1, our tool generates the witness `{"x": 0}`, as any valid instance

must be an object, where member x is mandatory and integer-typed. For Example 1.2, our tool generates the witness `{}`, since the empty object is valid. For the schema designer, this valuable feedback may well increase the overall productivity.

Moreover, upon the push of a button, the designer can generate further witnesses. In the example just discussed, the designer would be provided with `{"0": null}`. Thus, by interactive iteration, the designer ensures that he or she "gets it right".

Moreover, our tool allows the comparison of two schemas: for a witness that is valid w.r.t. the schema from Example 1.2, but not w.r.t. the schema from Example 1.1, the tool returns the JSON document `{}`. For the other way round, a witness is `{"x": 0}`. Again, the designer can request further witnesses, as needed.

Contributions. The goal of this demonstration is to showcase a tool allowing the schema designer to investigate the formal properties of a schema, and even to compare schemas. Our tool is based on our earlier contributions on algebraic manipulations of JSON Schema [7]. With our tool, the designer can:

- obtain an algebraic representation of the input schema;
- generate a witness for the schema, to verify whether the schema is empty or not, and to gain insights into the actual semantics of a given schema;
- exploit witness generation for checking whether a schema S_1 is a subtype of a schema S_2, and hence, whether it represents a conservative and not disruptive evolution.

This combination of features is currently not supported by any existing commercial or academic tool: while tools for JSON Schema containment checking are available (e.g., [9]), they merely produce boolean answers. Ours is the first tool capable of generating actual witnesses in containment checking.

2 JSON AND JSON SCHEMA

In the following, we introduce the JSON data model and provide some intuition for JSON Schema. We refer to [7, 10] for details.

2.1 JSON data model

The grammar below captures the syntax of JSON values, which are either basic values, objects, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Objects O represent sets of members, each member being a name-value pair, and arrays A.

$J ::=$	$B \mid O \mid A$		**JSON expressions**
$B ::=$	null $\mid$ true $\mid$ false $\mid n \mid s$		**Basic values**
	$n \in \mathsf{Num}, s \in \mathsf{Str}$		
$O ::=$	$\{l_1 : J_1, \ldots, l_n : J_n\}$		**Objects**
	$n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$		
$A ::=$	$[J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

2.2 JSON Schema

JSON Schema is a language for defining the structure of JSON documents, maintained by the Internet Engineering Task Force [4].

JSON Schema uses the JSON syntax. Each construct is defined using a JSON object with a set of fields describing assertions relevant for the values being described. Some assertions can be applied to any JSON value type (e.g., *type*), while others are more specific (e.g., *multipleOf* applies to numeric values only). The syntax and semantics of JSON Schema have been formalized in [10] following the specification of Draft-04. We limit ourself to an informal discussion about the possible constraints associated to each type:

- when defining a *string*, it is possible to restrict its length by specifying *minLength* and *maxLength* constraints, and to define the *pattern* that the string should match;
- when defining a *number*, it is possible to define its range of values (by any combination of *minimum* / *exclusiveMinimum* and *maximum* / *exclusiveMaximum*), and to define whether it should be *multipleOf* a given number;
- when defining an *object*, it is possible to define its *properties*, the type of its *additionalProperties* and the type of the properties matching a given pattern (i.e., *patternProperties*). It is also possible to restrict the minimum and maximum number of properties using *minProperties* and *maxProperties*, and to indicate which properties are *required*;
- when defining an *array*, it is possible to define the type of its *items* and the type of the *additionalItems* which were not already defined by *items*, and to restrict the minimum and maximum size of the array; moreover, it is also possible to enforce uniqueness of the items using *uniqueItems*.

JSON Schema allows the combination of assertions using standard boolean connectives: *not* for negation, *allOf* for conjunction, *anyOf* for disjunction, and *oneOf* for exclusive disjunction. A finite set of accepted values can be indicated through the *enum* constraint. Please note that hereafter, as well as in our formal development [7], we will use in some examples the usual notation for boolean operators (e.g., $\vee/\neg/\wedge$ for disjunction/negation/conjunction) and the symbols S/S_i to indicate schema fragments.

3 WITNESS GENERATION

Witness generation is challenging for JSON Schema, in particular due to the non-algebraic nature of JSON Schema (the meaning of certain assertions depends on the surrounding context), and because of the presence of negation and conjunctive schemas. We elaborate on these facts, and describe some main aspects of the formal systems and algorithms we devised. We will provide the full details in a future publication.

In a nutshell, our approach proceeds as follows. Assume that you have an algorithm to generate a witness for any schema assertion S of size up to n. In order to generate a witness for a schema of size $n + 1$ describing objects having a field of label l and value of type S, one will generate a witness w for S and use it to build an object with a field label l whose value is w.

For disjunction $S_1 \vee S_2$, we recursively generate witnesses of S_1 and of S_2. Yet negation and conjunction are problematic, as there is no way to generate a witness for $\neg S$ starting from a witness for S, and, given a witness for S_1, if this is not a witness for $S_1 \wedge S_2$, we may need to try infinitely many others before finding one that satisfies S_2. As we will see, since conjunction is used for object and array schemas, dealing with conjunction is particularly important for generating these two kinds of values.

Dealing with these challenges requires schema manipulations that can be rather complex. In order to devise the necessary schema transformation rules, as well as to study their properties and optimization techniques, we designed an algebra which is at the same time minimal and fully compliant to JSON Schema.

Details can be found in [7], but just to have a glimpse, consider the following JSON Schema fragment describing properties of label-value members of object values. In this fragment, we have a conjunction of assertions satisfied by a JSON value J if the following holds: if J is an object then i) if a k_i label is present,

then its associated value meets S_i, ii) if a label k' is present, satisfying a pattern (regular expression) r_i, then its associated value satisfies PS_i, iii) for all other labels in J not satisfying the previous conditions, the associated value satisfies S, iv) a member with label k_1 is required.

$$\begin{array}{l}\texttt{"properties"} : \{\texttt{"}k_1\texttt{"} : S_1, \ldots, \texttt{"}k_n\texttt{"} : S_n\},\\ \texttt{"patternProperties"} : \{\texttt{"}r_1\texttt{"} : PS_1, \ldots, \texttt{"}r_m\texttt{"} : PS_m\},\\ \texttt{"required"} : [\texttt{"}k_1\texttt{"}],\\ \texttt{"additionalProperties"} : S\end{array}$$

In this example, we can see the non-algebraic nature of JSON Schema: the semantics of one assertion (`additionalProperties`) depends on a co-occurring assertion (`properties`).

Our algebra encapsulates possibly interacting assertions into one, as shown below.

$$\begin{array}{l}\mathsf{props}(\underline{k_1} : \langle S_1\rangle, \ldots, \underline{k_n} : \langle S_n\rangle, r_1 : \langle PS_1\rangle, \ldots, r_m : \langle PS_m\rangle; \langle S\rangle)\\ \qquad \wedge\ \mathsf{req}(k_1)\end{array}$$

In the above algebra expression, we use $\underline{k}$ to indicate the JSON pattern "^$k$$" that only matches k, and $\langle S'\rangle$ to indicate our algebra expression corresponding to a schema S'.

By relying on this algebra, in order to deal with schemas $\neg S$ for enabling witness generation, we follow a traditional approach: we push negation inside S by playing with standard boolean laws, in order to obtain an equivalent, not-free schema that we use for witness generation. Unfortunately, JSON Schema does not enjoy negation closure: there are JSON schemas for which not-elimination is not possible. So we have extended our algebra with several new basic operators ensuring negation closure (that can be produced by not-elimination), and for which witness generation is possible in an inductive fashion, after further rewritings that we are going to exemplify. One of such operators is

$$\mathsf{pattReq}(r_1 : S_1, \ldots, r_n : S_n)$$

In order for a value to be an instance of the schema above, if the instance is an object, then, for each $i \in \{1..n\}$, it must possess a member whose name matches r_i and whose value satisfies S_i. (It is worth observing, that it is strictly more expressive than `required` since it allows one to require a name that belongs to an infinite set $L(r_i)$, and it associates a schema S_i to each required pattern r_i.)

A second challenging aspect is related to conjunction. In order to deal with conjunctive schemas we rely on standard rewritings, enabling the transformation into equivalent schemas in Disjoint Normal Form, which is more amenable for witness generation.

Unfortunately, DNF rewriting is not sufficient, because some mutual dependencies still remain among factors of conjunctions after DNF transformations. This means that we need to effectively push DNF transformation (as well as other operators) a step further in an unconventional fashion. The approach we have devised can be illustrated by the following example, where we focus on object schemas.

We use here JSON regular expressions (patterns), where $\hat{}$ matches the beginning of a string, $[\hat{}abc]$ matches any one character different from a, b and c, a dot . matches any character, \$ matches the end of the string, so that "$\hat{}a[\hat{}b]$." matches *acccccc* and *acc* but does not match *ac*, because the dot after the "$\hat{}a[\hat{}b]$" requires a third letter (carefully consider the dots in the patterns).

The expression below is a conjunction that we obtain by means of DNF transformations. We use the notation $\{\mathrm{Obj}, S_1, \ldots, S_n\}$ to denote a *group* of statements whose conjunction $\mathrm{Obj} \wedge S_1, \wedge \ldots \wedge S_n$ describes object values (we can also have array groups, etc.). Also note that $\mathbf{t}$ stands for the schema accepting any value.

$$\begin{array}{c}\{\mathrm{Obj}, \mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2),\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}), \mathsf{pattReq}(\hat{}\,a : S_3)\}\end{array}$$

A possible plausible witness generation strategy for this group would start considering pattReq constraints, but we need to keep into consideration possible interactions with other patterns in the object type, so we should first generate a witness for pattReq constraints, then checking whether props() are satisfied by the candidate witness, and if it is not the case, go back to pattReq and so on, by possibly infinite loops. To avoid this we rather manipulate the object group in order to be able to focus on subexpressions of the newly obtained object schema where, in some sense, all possible interactions are finitely enumerated, so that they can be dealt with separately.

Rather than providing the step-by-step process that produces this expansion, we show below the final result.

$$\begin{array}{l}\mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2) \rightarrow\\ \qquad \hat{}\,a[\hat{}b] : S_1, \hat{}\,ab : S_1 \wedge S_2, \hat{}\,[\hat{}a]b : S_2, \hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,ad : S_1 \wedge \neg S_3, \hat{}\,[\hat{}a]d : \mathbf{t})\\ \mathsf{pattReq}(\hat{}\,a : S_3) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,a[\hat{}bd] : S_1 \wedge S_3,\\ \qquad\qquad\qquad\qquad \hat{}\,ab : S_1 \wedge S_2 \wedge S_3),\end{array}$$

In the props()-part the set $\{\hat{}\,a, \hat{}\,.b\}$ has been divided into three disjoint parts $\{\hat{}\,a[\hat{}b], \hat{}\,ab, \hat{}\,[\hat{}a]b\}$ by separating the intersection $\hat{}\,ab$ from the two original patterns, and the set is completed with $\hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}$. Note that these new patterns can be obtained by means of standard techniques, thanks to the well-known closure properties of regular expressions.

The first request pattReq($\hat{}\,.d : \mathbf{t}$) is split into three different cases. The first $\hat{}\,ad : S_1 \wedge S_3$ is in common with the other orPattReq (an internal operator introduced to decompose pattReq into disjoint components), while the case $\hat{}\,ad : S_1 \wedge \neg S_3$ is internally and externally split, thanks to the $\neg S_3$ factor in the schema, and $\hat{}\,[\hat{}a]d$ is pattern-disjoint thanks to the initial $[\hat{}a]$. You can also observe that $\hat{}\,ad : S_1 \wedge S_3$ *internalizes* the requirement $\hat{}\,a : S_1$, the same holds for $\hat{}\,ad : S_1 \wedge \neg S_3$, while $\hat{}\,[\hat{}a]d$ only matches the trivial requirement, hence maintains its $\mathbf{t}$ schema.[1]

The second pattReq is split into three cases as well, in order to bring into view the intersection with the first pattReq, and in order to internalize the constraints of the props()-part.

This splitting effort is needed in order to be able to enumerate and try all the possible ways of satisfying a set of requests. For example, in this case the two orPattReq requests share the first component $\hat{}\,ad : S_1 \wedge S_3$, and contain two more components each, all of them mutually incompatible, hence having a structure $\mathsf{orPattReq}(a, b1, b2), \mathsf{orPattReq}(a, c1, c2)$. Hence, we know that there are exactly 5 ways of satisfying both: either by generating a single member that satisfies a, or by generating two members that satisfy, respectively, $(b1, c1)$, $(b1, c2)$, $(b2, c1)$, $(b2, c2)$, and our witness generation algorithm will try to pursue all, and only, these five approaches.

Even array groups obtained by DNF rewriting need preparation, by a different approach, which we cannot detail here, for

[1] As we have introduced not-schemas, notably $\neg S_3$, we re-apply not-elimination. For space reasons we do not delve into these aspects.

space reasons. Also, we have omitted how we deal with recursive definitions, both in not-elimination and witness generation.

This algorithm has an overall exponential complexity. However, we have designed techniques that make the problems tractable for many real-world schemas, and we are currently measuring their effectiveness.

4 DEMONSTRATION OVERVIEW

Our demo setup includes these datasets: we explore schemas from the JSON Schema Test Suite [2], a collection of small schemas that serve as unit tests for JSON Schema validators (and explore different operators), and real-world schemas from SchemaStore.org [5]. Naturally, our attendees may also formulate their own schemas.

We next describe the analysis for single schemas in more detail, and then remark on how attendees may also compare schemas.

Witness generation. Our tool is implemented as a Spring web application, with a Java backend. Our prototype does not yet support the operators `uniqueItems` and repeatedItems. Figure 1 shows a screenshot of the analysis of a single schema.

Typically, the user will first enter a JSON Schema document (or load one of the provided schemas), and then convert the schema (shown in the midsection of our screenshot) into our algebra (shown in the bottom section). Our algebra has been designed to be close to the original language, to be intuitive for practitioners. Yet different from the JSON Schema language, our algebra enjoys substitutability, that is, the semantics of an operator does not depend on its context, which eases manipulation.

The user may then choose to generate a first JSON witness. If the system finds no witness, it will alert the user that the schema is empty, otherwise, a witness is generated.

If there is a witness, the user can generate a further ("yet another") witness, that is different from all those previously seen. Alternatively, the user can edit the original JSON Schema, or directly the algebraic expression, and request that a new first witness is generated (disregarding witnesses already seen).

The schema designer can choose to convert back from the algebra to JSON Schema. Thus, the schema designer can interactively explore the semantics of a given schema, switch between the JSON Schema representation and the (often more compact) representation in our algebra, and iteratively revise the schema.

To allow interested demo attendees to inspect the internals of witness generation, as outlined in the previous section, our tool can also perform negation elimination on algebraic expressions. This feature would not be included in a tool targeted at end users.

Comparing schemas. Our tool offers a second screen (not shown here) where two schemas may be compared. Rather than computing a boolean answer to the question whether one schema subsumes the other, as done in state-of-the-art tools today [8], our tool can generate a witness that exemplifies a JSON document which is valid w.r.t. the one schema, but not the other.

Target audience. Our demo targets both the EDBT and the ICDT community. Attendees will become sensitive to the intricacies of working with the JSON Schema language, which caters to the ICDT community. Moreover, we point out original research questions that are of interest to the EDBT community, such as efficiency and scalability issues in dealing with real-world schemas, either due to the conditional semantics of the JSON Schema language, the interplay between negation and recursion (known to be difficult also in other areas of database research), and the sheer size of some real-world schemas (especially generated schemas, which can even take up hundreds of thousands of lines [8]).

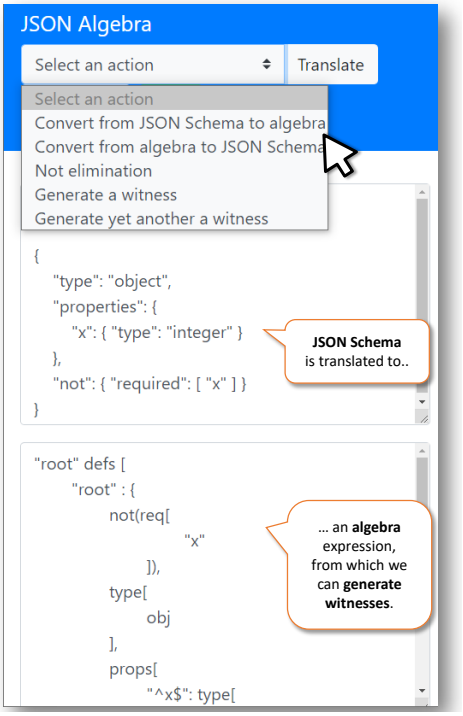

Figure 1: Screenshot: from JSON Schema to our algebra.

ACKNOWLEDGMENTS

Giorgio Ghelli's contribution has been funded by MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems). Stefanie Scherzinger's contribution has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 385808805.

REFERENCES

[1] [n.d.]. JSON Schema – valid if object does *not* contain a particular property. Available at:https://stackoverflow.com/questions/30515253/json-schema-valid-if-object-does-not-contain-a-particular-property.
[2] [n.d.]. JSON Schema Test Suite. Available at:https://github.com/json-schema-org/JSON-Schema-Test-Suite.
[3] 2020. Apache Avro. http://avro.apache.org.
[4] 2020. Internet Engineering Task Force. Available at https://www.ietf.org.
[5] 2020. JSON Schema Schema. https://www.schemastore.org/json/.
[6] 2020. MongoDB. https://www.mongodb.com.
[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Proc. BDA 2020.*
[8] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *Proc. EmpER.*
[9] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2020. Type Safety with JSON Subschema. arXiv:cs.PL/1911.12651
[10] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proc. WWW.* 263–273.
[11] Ion Stoica. 2020. Systems and ML: When the Sum is Greater than Its Parts. In *Proc. SIGMOD.*
[12] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02

CONTENTS

A Tool for 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

Francesco Falleni
Dipartimento di Informatica, Università di Pisa
fallenifrancesco98@gmail.com

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa
ghelli@di.unipi.it

Cristiano Landi
Dipartimento di Informatica, Università di Pisa
c.landi7@studenti.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

Stefanie Scherzinger
Universität Passau
stefanie.scherzinger@uni-passau.de

ABSTRACT

JSON Schema is an evolving standard for the description of JSON documents. It is an extremely powerful language endowed with boolean operators and recursive definitions. Hence, classical problems like schema *consistency* and *equivalence* may be challenging without well-principled tools. Based on our recent effort for laying down an algebraic formal semantics of JSON Schema, we demonstrate an approach for generating valid *witnesses* of a user-defined schema. Our goal is not only to allow programmers to design schemas that meet their intentions, but also to guide them in their journey to understanding the semantics of existing schemas, in an interactive fashion. We thus aim to contribute to the adoption of the JSON Schema language by facilitating its use.

1 INTRODUCTION

In recent years, JSON has become the *de facto* standard data interchange format, and is now widely used for exchanging data between web applications and remote servers, for exporting and importing data, as well as inside complex ML pipelines for combining different stages, as in Google TFX [11].

Despite its great popularity, there is no consensus about a *standard* schema language for JSON yet. Indeed, in many cases, JSON datasets come without a schema, and the end user or application has the duty to infer or guess a new schema, if required. In many other cases, however, several and vastly different schema languages are used for describing the structure of JSON data, ranging from Apache Avro [3], to the MongoDB internal schema language [6], and to JSON Schema [12].

Differently from what happened with XML, whose standard schema languages (DTDs and XML Schema) reached quickly a wide diffusion, JSON Schema is not being adopted at the same pace. Many reasons are slowing down its adoption, but, according to our observations, a major obstacle is the fact that, while extremely powerful, JSON Schema is – frankly – hard to use. Indeed, a schema is a logical combination of implicative assertions, and some of them may produce side effects on previous ones.

As a consequence, leaving the realm of plain vanilla schemas may expose the programmer to many risks, such as the definition of a schema with unintended semantics, or one that is even empty.

Example 1.1. Consider the following schema.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "required": [ "x" ]
}
```

This schema declares that all instances are JSON objects, and that each object has a mandatory *member* whose name is x and whose type is *integer*. This schema, however, does not impose further constraints on object values; therefore, an object may also have supplementary and unconstrained members.

Example 1.2. Consider now the following schema, differing only in the next-to-last line.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "not": { "required": [ "x" ] }
}
```

One may assume that this specifies that x is "not required", hence is optional. However, given the semantics of JSON Schema, negating a required member does not make it optional: indeed, the final effect is to actually forbid the presence of the member, hence excluding any JSON object having a member whose name is x (this example is inspired by a discussion on Stack Overflow [1], where the confusing effect of this schema is testified).

Given the complex and non-trivial interplay between schema assertions, designing a rich yet sound schema is challenging, especially when other powerful mechanisms of JSON Schema are involved, such as negation, mutual exclusion, recursion, union and conjunction, as well as array constraints controlling array length and content, possibly requiring uniqueness of elements.

Motivating Witness Generation. The state-of-the-art approach for exploring JSON Schema semantics is ultimately a manual trial and error: using a JSON Schema validator, a schema designer can test whether a JSON document is valid w.r.t. the schema. That is, the designer must come up with suitable *witness* documents.

Yet, in this demo, we present a tool capable of automatic witness generation. For instance, for the schema from Example 1.1, our tool generates the witness `{"x": 0}`, as any valid instance

must be an object, where member x is mandatory and integer-typed. For Example 1.2, our tool generates the witness `{}`, since the empty object is valid. For the schema designer, this valuable feedback may well increase the overall productivity.

Moreover, upon the push of a button, the designer can generate further witnesses. In the example just discussed, the designer would be provided with `{"0": null}`. Thus, by interactive iteration, the designer ensures that he or she "gets it right".

Moreover, our tool allows the comparison of two schemas: for a witness that is valid w.r.t. the schema from Example 1.2, but not w.r.t. the schema from Example 1.1, the tool returns the JSON document `{}`. For the other way round, a witness is `{"x": 0}`. Again, the designer can request further witnesses, as needed.

Contributions. The goal of this demonstration is to showcase a tool allowing the schema designer to investigate the formal properties of a schema, and even to compare schemas. Our tool is based on our earlier contributions on algebraic manipulations of JSON Schema [7]. With our tool, the designer can:

- obtain an algebraic representation of the input schema;
- generate a witness for the schema, to verify whether the schema is empty or not, and to gain insights into the actual semantics of a given schema;
- exploit witness generation for checking whether a schema S_1 is a subtype of a schema S_2, and hence, whether it represents a conservative and not disruptive evolution.

This combination of features is currently not supported by any existing commercial or academic tool: while tools for JSON Schema containment checking are available (e.g., [9]), they merely produce boolean answers. Ours is the first tool capable of generating actual witnesses in containment checking.

2 JSON AND JSON SCHEMA

In the following, we introduce the JSON data model and provide some intuition for JSON Schema. We refer to [7, 10] for details.

2.1 JSON data model

The grammar below captures the syntax of JSON values, which are either basic values, objects, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Objects O represent sets of members, each member being a name-value pair, and arrays A.

$J ::=$	$B \mid O \mid A$		**JSON expressions**
$B ::=$	null $\mid$ true $\mid$ false $\mid n \mid s$		**Basic values**
	$n \in \mathsf{Num}, s \in \mathsf{Str}$		
$O ::=$	$\{l_1 : J_1, \ldots, l_n : J_n\}$		**Objects**
	$n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$		
$A ::=$	$[J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

2.2 JSON Schema

JSON Schema is a language for defining the structure of JSON documents, maintained by the Internet Engineering Task Force [4].

JSON Schema uses the JSON syntax. Each construct is defined using a JSON object with a set of fields describing assertions relevant for the values being described. Some assertions can be applied to any JSON value type (e.g., *type*), while others are more specific (e.g., *multipleOf* applies to numeric values only). The syntax and semantics of JSON Schema have been formalized in [10] following the specification of Draft-04. We limit ourself to an informal discussion about the possible constraints associated to each type:

- when defining a *string*, it is possible to restrict its length by specifying *minLength* and *maxLength* constraints, and to define the *pattern* that the string should match;
- when defining a *number*, it is possible to define its range of values (by any combination of *minimum* / *exclusiveMinimum* and *maximum* / *exclusiveMaximum*), and to define whether it should be *multipleOf* a given number;
- when defining an *object*, it is possible to define its *properties*, the type of its *additionalProperties* and the type of the properties matching a given pattern (i.e., *patternProperties*). It is also possible to restrict the minimum and maximum number of properties using *minProperties* and *maxProperties*, and to indicate which properties are *required*;
- when defining an *array*, it is possible to define the type of its *items* and the type of the *additionalItems* which were not already defined by *items*, and to restrict the minimum and maximum size of the array; moreover, it is also possible to enforce uniqueness of the items using *uniqueItems*.

JSON Schema allows the combination of assertions using standard boolean connectives: *not* for negation, *allOf* for conjunction, *anyOf* for disjunction, and *oneOf* for exclusive disjunction. A finite set of accepted values can be indicated through the *enum* constraint. Please note that hereafter, as well as in our formal development [7], we will use in some examples the usual notation for boolean operators (e.g., $\vee/\neg/\wedge$ for disjunction/negation/conjunction) and the symbols S/S_i to indicate schema fragments.

3 WITNESS GENERATION

Witness generation is challenging for JSON Schema, in particular due to the non-algebraic nature of JSON Schema (the meaning of certain assertions depends on the surrounding context), and because of the presence of negation and conjunctive schemas. We elaborate on these facts, and describe some main aspects of the formal systems and algorithms we devised. We will provide the full details in a future publication.

In a nutshell, our approach proceeds as follows. Assume that you have an algorithm to generate a witness for any schema assertion S of size up to n. In order to generate a witness for a schema of size $n + 1$ describing objects having a field of label l and value of type S, one will generate a witness w for S and use it to build an object with a field label l whose value is w.

For disjunction $S_1 \vee S_2$, we recursively generate witnesses of S_1 and of S_2. Yet negation and conjunction are problematic, as there is no way to generate a witness for $\neg S$ starting from a witness for S, and, given a witness for S_1, if this is not a witness for $S_1 \wedge S_2$, we may need to try infinitely many others before finding one that satisfies S_2. As we will see, since conjunction is used for object and array schemas, dealing with conjunction is particularly important for generating these two kinds of values.

Dealing with these challenges requires schema manipulations that can be rather complex. In order to devise the necessary schema transformation rules, as well as to study their properties and optimization techniques, we designed an algebra which is at the same time minimal and fully compliant to JSON Schema.

Details can be found in [7], but just to have a glimpse, consider the following JSON Schema fragment describing properties of label-value members of object values. In this fragment, we have a conjunction of assertions satisfied by a JSON value J if the following holds: if J is an object then i) if a k_i label is present,

then its associated value meets S_i, ii) if a label k' is present, satisfying a pattern (regular expression) r_i, then its associated value satisfies PS_i, iii) for all other labels in J not satisfying the previous conditions, the associated value satisfies S, iv) a member with label k_1 is required.

$$\begin{array}{l}\texttt{"properties"} : \{\texttt{"}k_1\texttt{"} : S_1, \ldots, \texttt{"}k_n\texttt{"} : S_n\},\\ \texttt{"patternProperties"} : \{\texttt{"}r_1\texttt{"} : PS_1, \ldots, \texttt{"}r_m\texttt{"} : PS_m\},\\ \texttt{"required"} : [\texttt{"}k_1\texttt{"}],\\ \texttt{"additionalProperties"} : S\end{array}$$

In this example, we can see the non-algebraic nature of JSON Schema: the semantics of one assertion (`additionalProperties`) depends on a co-occurring assertion (`properties`).

Our algebra encapsulates possibly interacting assertions into one, as shown below.

$$\begin{array}{l}\mathsf{props}(\underline{k_1} : \langle S_1\rangle, \ldots, \underline{k_n} : \langle S_n\rangle, r_1 : \langle PS_1\rangle, \ldots, r_m : \langle PS_m\rangle; \langle S\rangle)\\ \qquad \wedge\ \mathsf{req}(k_1)\end{array}$$

In the above algebra expression, we use $\underline{k}$ to indicate the JSON pattern "^$k$$" that only matches k, and $\langle S'\rangle$ to indicate our algebra expression corresponding to a schema S'.

By relying on this algebra, in order to deal with schemas $\neg S$ for enabling witness generation, we follow a traditional approach: we push negation inside S by playing with standard boolean laws, in order to obtain an equivalent, not-free schema that we use for witness generation. Unfortunately, JSON Schema does not enjoy negation closure: there are JSON schemas for which not-elimination is not possible. So we have extended our algebra with several new basic operators ensuring negation closure (that can be produced by not-elimination), and for which witness generation is possible in an inductive fashion, after further rewritings that we are going to exemplify. One of such operators is

$$\mathsf{pattReq}(r_1 : S_1, \ldots, r_n : S_n)$$

In order for a value to be an instance of the schema above, if the instance is an object, then, for each $i \in \{1..n\}$, it must possess a member whose name matches r_i and whose value satisfies S_i. (It is worth observing, that it is strictly more expressive than `required` since it allows one to require a name that belongs to an infinite set $L(r_i)$, and it associates a schema S_i to each required pattern r_i.)

A second challenging aspect is related to conjunction. In order to deal with conjunctive schemas we rely on standard rewritings, enabling the transformation into equivalent schemas in Disjoint Normal Form, which is more amenable for witness generation.

Unfortunately, DNF rewriting is not sufficient, because some mutual dependencies still remain among factors of conjunctions after DNF transformations. This means that we need to effectively push DNF transformation (as well as other operators) a step further in an unconventional fashion. The approach we have devised can be illustrated by the following example, where we focus on object schemas.

We use here JSON regular expressions (patterns), where $\hat{}$ matches the beginning of a string, $[\hat{}abc]$ matches any one character different from a, b and c, a dot . matches any character, \$ matches the end of the string, so that "$\hat{}a[\hat{}b]$." matches *acccccc* and *acc* but does not match *ac*, because the dot after the "$\hat{}a[\hat{}b]$" requires a third letter (carefully consider the dots in the patterns).

The expression below is a conjunction that we obtain by means of DNF transformations. We use the notation $\{\mathrm{Obj}, S_1, \ldots, S_n\}$ to denote a *group* of statements whose conjunction $\mathrm{Obj} \wedge S_1, \wedge \ldots \wedge S_n$ describes object values (we can also have array groups, etc.). Also note that $\mathbf{t}$ stands for the schema accepting any value.

$$\begin{array}{c}\{\mathrm{Obj}, \mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2),\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}), \mathsf{pattReq}(\hat{}\,a : S_3)\}\end{array}$$

A possible plausible witness generation strategy for this group would start considering pattReq constraints, but we need to keep into consideration possible interactions with other patterns in the object type, so we should first generate a witness for pattReq constraints, then checking whether props() are satisfied by the candidate witness, and if it is not the case, go back to pattReq and so on, by possibly infinite loops. To avoid this we rather manipulate the object group in order to be able to focus on subexpressions of the newly obtained object schema where, in some sense, all possible interactions are finitely enumerated, so that they can be dealt with separately.

Rather than providing the step-by-step process that produces this expansion, we show below the final result.

$$\begin{array}{l}\mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2) \rightarrow\\ \qquad \hat{}\,a[\hat{}b] : S_1, \hat{}\,ab : S_1 \wedge S_2, \hat{}\,[\hat{}a]b : S_2, \hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,ad : S_1 \wedge \neg S_3, \hat{}\,[\hat{}a]d : \mathbf{t})\\ \mathsf{pattReq}(\hat{}\,a : S_3) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,a[\hat{}bd] : S_1 \wedge S_3,\\ \qquad\qquad\qquad\qquad \hat{}\,ab : S_1 \wedge S_2 \wedge S_3),\end{array}$$

In the props()-part the set $\{\hat{}\,a, \hat{}\,.b\}$ has been divided into three disjoint parts $\{\hat{}\,a[\hat{}b], \hat{}\,ab, \hat{}\,[\hat{}a]b\}$ by separating the intersection $\hat{}\,ab$ from the two original patterns, and the set is completed with $\hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}$. Note that these new patterns can be obtained by means of standard techniques, thanks to the well-known closure properties of regular expressions.

The first request pattReq($\hat{}\,.d : \mathbf{t}$) is split into three different cases. The first $\hat{}\,ad : S_1 \wedge S_3$ is in common with the other orPattReq (an internal operator introduced to decompose pattReq into disjoint components), while the case $\hat{}\,ad : S_1 \wedge \neg S_3$ is internally and externally split, thanks to the $\neg S_3$ factor in the schema, and $\hat{}\,[\hat{}a]d$ is pattern-disjoint thanks to the initial $[\hat{}a]$. You can also observe that $\hat{}\,ad : S_1 \wedge S_3$ *internalizes* the requirement $\hat{}\,a : S_1$, the same holds for $\hat{}\,ad : S_1 \wedge \neg S_3$, while $\hat{}\,[\hat{}a]d$ only matches the trivial requirement, hence maintains its $\mathbf{t}$ schema.[1]

The second pattReq is split into three cases as well, in order to bring into view the intersection with the first pattReq, and in order to internalize the constraints of the props()-part.

This splitting effort is needed in order to be able to enumerate and try all the possible ways of satisfying a set of requests. For example, in this case the two orPattReq requests share the first component $\hat{}\,ad : S_1 \wedge S_3$, and contain two more components each, all of them mutually incompatible, hence having a structure $\mathsf{orPattReq}(a, b1, b2), \mathsf{orPattReq}(a, c1, c2)$. Hence, we know that there are exactly 5 ways of satisfying both: either by generating a single member that satisfies a, or by generating two members that satisfy, respectively, $(b1, c1)$, $(b1, c2)$, $(b2, c1)$, $(b2, c2)$, and our witness generation algorithm will try to pursue all, and only, these five approaches.

Even array groups obtained by DNF rewriting need preparation, by a different approach, which we cannot detail here, for

[1] As we have introduced not-schemas, notably $\neg S_3$, we re-apply not-elimination. For space reasons we do not delve into these aspects.

space reasons. Also, we have omitted how we deal with recursive definitions, both in not-elimination and witness generation.

This algorithm has an overall exponential complexity. However, we have designed techniques that make the problems tractable for many real-world schemas, and we are currently measuring their effectiveness.

4 DEMONSTRATION OVERVIEW

Our demo setup includes these datasets: we explore schemas from the JSON Schema Test Suite [2], a collection of small schemas that serve as unit tests for JSON Schema validators (and explore different operators), and real-world schemas from SchemaStore.org [5]. Naturally, our attendees may also formulate their own schemas.

We next describe the analysis for single schemas in more detail, and then remark on how attendees may also compare schemas.

Witness generation. Our tool is implemented as a Spring web application, with a Java backend. Our prototype does not yet support the operators `uniqueItems` and repeatedItems. Figure 1 shows a screenshot of the analysis of a single schema.

Typically, the user will first enter a JSON Schema document (or load one of the provided schemas), and then convert the schema (shown in the midsection of our screenshot) into our algebra (shown in the bottom section). Our algebra has been designed to be close to the original language, to be intuitive for practitioners. Yet different from the JSON Schema language, our algebra enjoys substitutability, that is, the semantics of an operator does not depend on its context, which eases manipulation.

The user may then choose to generate a first JSON witness. If the system finds no witness, it will alert the user that the schema is empty, otherwise, a witness is generated.

If there is a witness, the user can generate a further ("yet another") witness, that is different from all those previously seen. Alternatively, the user can edit the original JSON Schema, or directly the algebraic expression, and request that a new first witness is generated (disregarding witnesses already seen).

The schema designer can choose to convert back from the algebra to JSON Schema. Thus, the schema designer can interactively explore the semantics of a given schema, switch between the JSON Schema representation and the (often more compact) representation in our algebra, and iteratively revise the schema.

To allow interested demo attendees to inspect the internals of witness generation, as outlined in the previous section, our tool can also perform negation elimination on algebraic expressions. This feature would not be included in a tool targeted at end users.

Comparing schemas. Our tool offers a second screen (not shown here) where two schemas may be compared. Rather than computing a boolean answer to the question whether one schema subsumes the other, as done in state-of-the-art tools today [8], our tool can generate a witness that exemplifies a JSON document which is valid w.r.t. the one schema, but not the other.

Target audience. Our demo targets both the EDBT and the ICDT community. Attendees will become sensitive to the intricacies of working with the JSON Schema language, which caters to the ICDT community. Moreover, we point out original research questions that are of interest to the EDBT community, such as efficiency and scalability issues in dealing with real-world schemas, either due to the conditional semantics of the JSON Schema language, the interplay between negation and recursion (known to be difficult also in other areas of database research), and the sheer size of some real-world schemas (especially generated schemas, which can even take up hundreds of thousands of lines [8]).

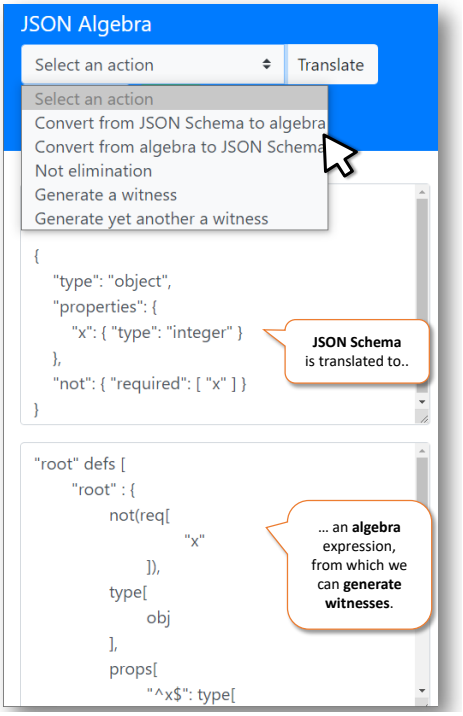

Figure 1: Screenshot: from JSON Schema to our algebra.

ACKNOWLEDGMENTS

Giorgio Ghelli's contribution has been funded by MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems). Stefanie Scherzinger's contribution has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 385808805.

REFERENCES

[1] [n.d.]. JSON Schema – valid if object does *not* contain a particular property. Available at:https://stackoverflow.com/questions/30515253/json-schema-valid-if-object-does-not-contain-a-particular-property.
[2] [n.d.]. JSON Schema Test Suite. Available at:https://github.com/json-schema-org/JSON-Schema-Test-Suite.
[3] 2020. Apache Avro. http://avro.apache.org.
[4] 2020. Internet Engineering Task Force. Available at https://www.ietf.org.
[5] 2020. JSON Schema Schema. https://www.schemastore.org/json/.
[6] 2020. MongoDB. https://www.mongodb.com.
[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Proc. BDA 2020.*
[8] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *Proc. EmpER.*
[9] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2020. Type Safety with JSON Subschema. arXiv:cs.PL/1911.12651
[10] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proc. WWW.* 263–273.
[11] Ion Stoica. 2020. Systems and ML: When the Sum is Greater than Its Parts. In *Proc. SIGMOD.*
[12] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02