Not Elimination and Witness Generation for JSON Schema (short version)

Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger

Not Elimination and Witness Generation for JSON Schema (short version)

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

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

Giorgio Ghelli
Dipartimento di Informatica,
Università di Pisa
ghelli@di.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 families of JSON documents. JSON Schema is a logical language, based on a set of *assertions* that describe features of the JSON value under analysis and on logical or structural combinators for these assertions. As for any logical language, problems like satisfaction, not-elimination, schema satisfiability, schema inclusion and equivalence, as well as witness generation, have both theoretical and practical interest. While satisfaction is trivial, all other problems are quite difficult, due to the combined presence of negation, recursion, and complex assertions in JSON Schema. To make things even more complex and interesting, JSON Schema is not algebraic, since we have both syntactic and semantic interactions between different keywords in the same schema object.

With such motivations, we present in this paper an algebraic characterization of JSON Schema, obtained by adding opportune operators, and by mirroring existing ones. We present then algebra-based approaches for dealing with not-elimination and witness generation problems, which play a central role as they lead to solutions for the other mentioned complex problems.

KEYWORDS

JSON Schema, negation, witness generation

1 INTRODUCTION

JSON Schema [2] is an evolving standard for the description of families of JSON documents. It is maintained by the Internet Engineering Task Force IETF [1]. Its latest version has been produced on 2019-09 [9] but is not widely used compared to the intermediate Draft-06.

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* that applies to numeric values only). The syntax and semantics of JSON Schema have been formalized in [8] following the specification of Draft-04. We limit ourself to an informal discussion revealing the possible constraints associated to each kind of type:

- when defining a *string*, it is possible to restrict its length by specifying the *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 specifying 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 unicity of the items using *uniqueItems*.

JSON Schema is a logical language allowing for combining assertions using standard boolean connectives: *not* for negation, *allOf* for conjunction, *anyOf* for disjunction, and *oneOf* for exclusive disjunction. As for any logical language, the following problems have a theoretical and practical interest:

- satisfaction $J \vDash S$: does a JSON document J satisfy schema S?
- not-elimination: is it possible to rewrite a schema to an equivalent form without negation?
- satisfiability of a schema: does a document J exist such that $J \vDash S$?
- schema inclusion $S \subseteq S'$: does, for each document J, $J \vDash S \Rightarrow J \vDash S'$?
- schema equivalence $S \equiv S'$: does, for each document J, $J \vDash S \Leftrightarrow J \vDash S'$?
- witness generation: is there an algorithm to generate one element J for any non-empty schema S?

While satisfaction is trivial, all other problems are quite difficult, due to the combined presence of negation, recursion, and complex assertions.

A second aspect that makes the task difficult is the non-algebraic nature of JSON Schema. A language is "algebraic" when the applicability and the semantics of its operators only depends on the semantics of their operands. In this sense, JSON Schema is not algebraic, since we have both syntactic and semantic interactions between different keywords in the same schema object, such as the prohibition to repeat a keyword inside a schema object, or the interactions between the "properties" and "additionalProperties" keywords. For instance, the following schema[1] demands that any properties other than foo and bar must have boolean values.

```
{ "properties": {"foo": {}, "bar": {}},
  "additionalProperties": {"type": "boolean"} }
```

Such features complicate the tasks of reasoning about the language and of writing code for its manipulation.

2 SUMMARY OF CONTRIBUTIONS

JSON Algebra. We define a *core algebra*, which features a subset of JSON Schema assertions. This algebra is minimal, that is, no operator can be defined starting from the others.

Not elimination. We show that negation cannot be eliminated from JSON Schema, since there are some assertions whose complement cannot be expressed without negation such as uniqueItems or multipleOf. We enrich the core algebra with primitive operators to express those missing complementary operators, and we present a *not elimination* algorithm for the enriched algebra. To our knowledge, this is the first paper where not elimination is completely defined, with particular regard to the treatment of negation and recursion.

Witness generation. We define an approach for witness generation for the complete JSON Schema language, with the only exception of the uniqueItems operator, hence solving the satisfiability and inclusion problems for this sublanguage.

For space reasons, many details and formal aspects presented in the complete report [4] are not reported here, including the extension to uniqueItems for witness generations. The presentation of several steps (especially for witness generation) is driven/based by/on examples.

Also, we would like to stress that results presented in this paper takes part of research activities [4] that are still in progress. So our main aim here is to present existing results, mainly at the definition and formalisation level of algorithms.

3 RELATED WORK

The first effort to formalize the semantics of JSON Schema as by Pezoa et al. in [8] whose goal was to lay the foundations of the JSON schema proposal by studying its expressive power and the complexity of the validation problem. Along the lines of this work, Bouhris et al. [6] characterized the expressivity of the JSON Schema language and investigated the complexity of the satisfiability problem which turns out to be *2EXPTIME* in the general case and *EXPSPACE* when disallowing *uniqueItems*. None of the above works study the problem of generating an instance of a JSON Schema. The only attempt to solve this problem was investigated by Earle et. al [5] in the context of testing REST API calls but the presented solution, which is based on translating JSON Schema definitions into an Erlang expression, is not formally defined and restricted to atomic values, objects and to some form of boolean expressions.

From the point of view of schema normalization, the closest work to ours is the one in [7] which studies schema inclusion for JSON Schema. To cope with the high expressivity of the JSON Schema language, a pre-requisite step is needed to rewrite the schemas into a *Disjunctive Normal Form* which has some similarities with the preparation phase of our work. However, compared to our work, the schema normalization in [7] lacks the ability of eliminating negation for all kinds constraints, does not deal with recursive definitions and is not able to decide schema satisfiability which is captured by the *inhabited( )* predicate whose specification is only informally discussed. This has been confirmed in practice by experimenting the tool developed in [7] for parsing real world schemas described in [4]: the tool raised an issue for 21,859 out of 23,480 input schemas. The dominating error is related to constructs not being supported, but many other errors due to the inability to parse recursive schemas or to navigate references are present.

4 CONCLUSION

JSON Schema is an evolving standard for the description of families of JSON documents, and is widely used in data-centric applications. Despite the recent interest in the research community related to this schema language, crucial problems like schema equivalence/inclusion and consistency have either been partially dealt with or not explored at all. In this work we present our approach in order to solve these problems, based on our algebraic specification of JSON Schema. We are currently finalizing a Java implementation of the presented algorithm, and studying optimisation techniques, by analysing a large repository of JSON Schemas allowing us for determining how often mechanisms that are critical for execution times are used. We are also investigating witness generation techniques able to generate several instances meant be used for testing queries and programs manipulating valid JSON data.

ACKNOWLEDGEMENTS

The research has been partially supported by the MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems).

REFERENCES

[1] Internet engineering task force, 2020. Available at https://www.ietf.org.
[2] Json schema, 2020. Available at https://json-schema.org.
[3] Json schema test suite, 2020. https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/master/tests/draft6/additionalProperties.json.
[4] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Not elimination and witness generation for json schema. 2020. Available at https://webia.lip6.fr/~baazizi/rs/js/dism/witnessgen.pdf.
[5] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño. Jsongen: a quickcheck based library for testing json web services. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 33–41, 2014.
[6] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. JSON: data model, query languages and schema specification. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *PODS*, pages 123–135. ACM, 2017.
[7] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Type safety with json subschema, 2019.
[8] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *WWW '16*, pages 263–273, 2016.
[9] A. Wright, H. Andrews, and B. Hutton. JSON Schema validation: A vocabulary for structural validation of json - draft-handrews-json-schema-validation-02. Technical report, Internet Engineering Task Force, sep 2019.

[1] Example available at [3].