Satisfiability of JSON Schema and Instance Validation

Juan L. Reutter, Martin Ugarte, and Domagoj Vrgoč

PUC Chile

Abstract. We study the JSON schema specification that provides a way of describing sets of JSON documents. In this paper we formalize a fragment of the specification and show that for this fragment the satisfiability problem for JSON schemas is PSPACE-complete. We also comment on our current line of research in this respect, stating our immediate and long term goals.

1 Introduction

JSON (JavaScript Object Notation) is a lightweight data-interchange format, standarized by the European association for standardizing information and communication systems (ECMA) and the Internet Engineering Task Force (IETF) [2, 3]. The JSON definitions are based on Javascript notation, and since its introduction it has gained immense popularity amongst web and api developers.

With the popularity of JSON it was soon noted that in many scenarios one can benefit from a declarative way of specifying a *schema* for JSON documents: In the basic *API* scenario this would allow developers to specify what type of JSON documents are accepted as inputs by the API and what is the format of the output of this API. This gave way to several schema specifications for JSON documents. We focus here on one of them, called *JSON schema* [5].

While JSON schema is still a work in progress, there is a growing body of applications that support JSON schema definitions, and a good deal of tools and packages to enable the validation of documents against JSON schema. We are currently at the fourth version of the JSON schema specification [4], and we have seen how the definition is being established as the default schema specification for JSON documents. However, to the best of our knowledge there has been no formal study of the schema specifications, and the scientific community has not been involved in the design choices for this language.

Our goal is to start this formal study and use it to recommend algorithms or identify potential weaknesses of the specification to the JSON schema working group. In this paper we establish the first formal model of (a fragment of) JSON schema, and use it to devise an algorithm for the satisfiability problem for schemas. We also shed light on the future milestones of our project, as well as our expected future outcomes.

1.1 JSON Documents

Let Σ be an alphabet and `true`, `false` two symbols not in Σ. The set of JSON objects is recursively defined as follows:

1. `true` and `false` are JSON objects, called *boolean objects*.
2. A real number (e.g. 3.14, 23) is a JSON object, called a *number object*, or just number.
3. Any string over Σ is a JSON object, called a *string object* (or just string).
4. If $t_1, \ldots, t_n$ are JSON objects then $t = [t_1, \ldots, t_n]$ is a JSON object called an *array*. In this case $t_1, \ldots, t_n$ are called the *objects of* t. If all of $t_1 \ldots, t_n$ are of the same type $\mathcal{T}$, we say that t is a $\mathcal{T}$ array (e.g. a string array).
5. If $t_1, \ldots, t_n$ are valid JSON objects and $s_1, \ldots, s_n$ are pairwise distinct string objects, then $t = \{s_1 : t_1, \ldots, s_n : t_n\}$ is a JSON object, called a *dictionary object*, or just dictionary. In this case, each $s_i : t_i$ is called a key-object pair of t.

The following syntax is normally used to navigate through JSON documents. If t is a dictionary object, then t[“key”] is the set of all key-objects pairs of t whose key is the string “key”. Likewise, if t is an array, then $t[n]$, for a natural number n contains the i-th object of t. In both cases, $\mathit{len}(t)$ denotes the amount of key-object pairs or objects in t, respectively.

1.2 JSON Schema

The idea of a JSON schema is to specify what type of objects are allowed in the documents conforming to the schema, and what are the properties and shape of these objects. We focus on a subset of the JSON schema definition that defines strings and dictionaries, according to the following syntax:

$$\begin{aligned}
\text{obj} &::= \text{str} \mid \text{dict} \mid \text{complex}\\
\text{str} &::= \text{`}\{\texttt{type:string}\text{'}\ \text{pattern?}\ \text{`}\}\text{'}\\
\text{dict} &::= \text{`}\{\texttt{type:dict}\text{'}\ \text{prop?}\ \text{patPr?}\ \text{required?}\ \text{addPr?}\ \text{minPr?}\ \text{maxPr?}\ \text{`}\}\text{'}\\
\text{complex} &::= \text{`}\texttt{not:[}\text{'}\,\text{obj}\,\text{`}\texttt{]}\text{'} \mid \text{`}\texttt{anyOf:[}\text{'}\,\text{obj}^+\,\text{`}\texttt{]}\text{'} \mid \text{`}\texttt{allOf:[}\text{'}\,\text{obj}^+\,\text{`}\texttt{]}\text{'}
\end{aligned}$$

Pairs `type:string` and `type:dict` indicate that the object must be a string or a dictionary, respectively. The definition also allows for the definition of complex objects that state that the object must belong to the complement of a given schema, to the union of a set of schemas, or to the intersection of a set of schemas.

For string types the definition allows to state that the string must conform to a certain regular expression over the alphabet Σ. For dictionaries several options are provided, and are defined according to the following syntax

$$\begin{aligned}
\text{pattern} &::= \text{`}\texttt{pattern:}\text{'}\ \mathit{regexp}\\
\text{prop} &::= \text{`}\texttt{properties:}\{\text{'}\,w_1 : \text{obj}_1, \ldots, w_n : \text{obj}_n\,\text{`}\}\text{'}\\
\text{patPr} &::= \text{`}\texttt{patternProperties:}\{\text{'}\,\mathit{regexp}_1 : \text{obj}_1, \ldots, \mathit{regexp}_m : \text{obj}_n\,\text{`}\}\text{'}\\
\text{addPr} &::= \text{`}\texttt{additionalProperties:}\text{'}\ \text{obj}\\
\text{required} &::= \text{`}\texttt{required:[}\text{'}\,w_1, \ldots, w_n\,\text{`}\texttt{]}\text{'}\\
\text{minPr} &::= \text{`}\texttt{minProperties:}\text{'}\ n\\
\text{maxPr} &::= \text{`}\texttt{maxProperties:}\text{'}\ n
\end{aligned}$$

Here $w, w_1, \ldots, w_n$ are strings over Σ, *regexp*, *regexp*$_1, \ldots,$ *regexp*$_m$ are regular expressions over Σ and n is a natural number. Pairs of form "w_i : obj$_i$" and "*regexp*$_i$: obj$_i$" inside prop and patPr state that, should the key w_i or a key conforming to *regexp*$_i$ be in the dictionary, then the value of that pair must conform to the schema given by obj$_i$. Pair "`additionalProperties` : obj" states that the values of all keys not explicitly mentioned in prop or patPr need to conform to the schema of obj, minPr and maxPr specify the number of pairs in the dictionary and "`required` : $[w_1, \ldots, w_n]$" states that there must be pairs with keys $w_1, \ldots, w_n$.

Unfortunately, we do not have enough space to give any meaningful examples. We do refer the reader to [5] for examples. Next, we formally define the semantics of JSON schema.

Semantics. Let t be a json object and let S be a JSON Schema. We say that t conforms to S, denoted by $t \models S$ if $S = \texttt{true}$ or if

- S is "allOf:[r]" and t conforms to every JSON Schema in r.
- S is "anyOf:[r]" and t conforms to at least one JSON Schema in r.
- S is "not:[r]" and t does not conform to r.
- t is a string and every key-value pair $s : r$ in S is either "`type:string`" or "`'pattern:'` *regexp*" and t is in the language of *regexp*.
- t is a dictionary and every key-value pair $s : r$ in S is either:
  - "type : dict"
  - $s =$ "minProperties" and *len*(t) is greater than or equal to r, or
  - $s =$ "maxProperties" and *len*(t) is less than or equal to r, or
  - $s =$ "required" and all of the strings in r are keys in t.
  - $s =$ "properties", $r = \{s_1 : r_1, \ldots, s_n : r_n\}$ and for every key-value pair $k : v$ in t, if $k = s_i$ for some $1 \leq i \leq n$, then t conforms to r_i.
  - $s =$ "patternProperties", $r = \{s_1 : r_1, \ldots, s_n : r_n\}$ and for every key-value pair $k : v$ in t such that k is not a key in S[properties] or S[properties] is not defined, if k is in the language of s_i, for $1 \leq i \leq n$, then v conforms to r_i
  - $s =$ "additionalProperties" and for every pair $k : v$ in t such that (1) k is not a key in S[properties] or S[properties] is not defined and (2) k does not conform to the regular expression generated by any key in S[patternProperties] or S[patternProperties] is not defined it must be the case that t conforms to v.

2 Static Analyisis

The first question that we chose to study is satisfiability: Given a JSON schema S, does there exist a JSON document t such that $t \models S$? Our JSON schemas are not recursive, but the complex constructs give them quite a lot of expressive power, even if they can only define documents given by nested dictionaries and strings. Nevertheless, we can show that satisfiability can be decided in PSPACE for our types of schemas, which is really the least we can ask for, as the combination of 'not:", "anyOf:" and "pattern:" allows us to easily code containment or equivalence of regular expressions, a problem known to be PSPACE-hard.

Theorem 1. *The satisfiability problem for JSON schemas is* PSPACE*-complete*

Proof (Sketch). Since JSON schema definitions are not recursive, they are satisfied by JSON documents with a bounded number of nested dictionaries, namely the maximum nesting of dictionaries in the JSON schema. Let then S be a schema. For the proof we use a coding scheme similar to that of [1] to transform JSON documents into strings, taking into account the current depth of nesting within these objects. We then construct an automaton A_S that accepts all strings that represent encodings for JSON documents. Because of the complex constructs, we need to use an alternating finite automata [6], that can be complemented and intersected in polynomial time. The construction is defined inductively, complementing and intersecting these automata every time we encounter a complex object. Once the construction is done, we can show that S is satisfiable if and only if the language defined by A_S is nonempty, a decision problem known to be in PSPACE.

3 Current Work

We have defined here a strict subset of the features allowed in JSON schema. Two additional features that do not take us away from defining JSON objects based solely on string and dictionaries, are dependencies and definitions.

Dependencies are pairs of form "dependency:$\{w_1 : \text{obj}_1, \ldots, w_n : \text{obj}_n\}$", and their semantics is as follows: If t is a dictionary object, then t conforms to the dependency of the form above if for every pair $k : v$ in t, if $k = w_i$ for some $1 \leq i \leq n$, it is the case that t also conforms to obj_i. Definitions are complex objects that allow for recursion, they are again pairs of form "definitions:$\{w_1 : \text{obj}_1, \ldots, w_n : \text{obj}_n\}$", but having these definitions allows one to use w_1 to refer to the schema obj_1. But of course, nothing prevents the schema obj_1 to use itself copies of w_1, thus allowing users to create recursive schema definitions.

Our next immediate goal is to understand the satisfiability of JSON schemas and validation of documents with respect to the schemas extended with these additional constructs, and we already have preliminary results. We are also studying the problem of validation of JSON documents against a schema (this problem is in polynomial time for schemas presented here). By studying these problems we hope to gain enough understanding of what features are simple from a computational point of view, and which of them have the potential to cause problems in real implementations.

References

1. M. Benedikt, W. Fan, and F. Geerts. Xpath satisfiability in the presence of dtds. *Journal of the ACM (JACM)*, 55(2):8, 2008.
2. T. Bray. The javascript object notation (json) data interchange format. 2014.
3. D. Crockford. The json data interchange format. Technical report, Technical report, ECMA International, October, 2013.
4. F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.
5. json-schema.org: The home of json schema. http://json-schema.org/.
6. R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.