Foundations of JSON Schema

Felipe Pezoa
PUC Chile
fipezoa@uc.cl

Juan L. Reutter
PUC Chile
jreutter@ing.puc.cl

Fernando Suarez
PUC Chile
fsuarez1@ing.puc.cl

Martín Ugarte
Université Libre de Bruxelless
martin.ugarte@ulb.ac.be

Domagoj Vrgoč
PUC Chile
dvrgoc@ing.puc.cl

ABSTRACT

JSON { the most popular data format for sending API requests and responses { is still lacking a standardized schema or meta-data de nition that allows the developers to specify the structure of JSON documents. JSON Schema is an attempt to provide a general purpose schema language for JSON, but it is still work in progress, and the formal speci cation has not yet been agreed upon. Why this could be a problem becomes evident when examining the behaviour of numerous tools for validating JSON documents against this initial schema proposal: although they agree on most general cases, when presented with the greyer areas of the speci cation they tend to di er signi cantly. In this paper we provide the rst formal de nition of syntax and semantics for JSON Schema and use it to show that implementing this layer on top of JSON is feasible in practice. This is done both by analysing the theoretical aspects of the validation problem and by showing how to set up and validate a JSON Schema for Wikidata, the central storage for Wikimedia.

1. INTRODUCTION

JSON (JavaScript Object Notation) [5, 19, 9] is a data format based on the data types of the JavaScript programming language. In the last few years JSON has gained tremendous popularity among web developers, and has become the main format for exchanging information over the web.

JSON nowadays plays a key role in web applications. Indeed, software executing functions ordered by remote machines must establish a precise protocol for receiving and answering requests, which is called an Application Programming Interface (API). Given that JSON is a language which can be easily understood by both developers and machines, it has become the most popular format to send API requests and responses over the HTTP protocol. As an example, consider an application containing information about weather conditions around the world. The application provides an API to allow other software to access this information. A hypothetical call to this API coulds be a request containing this JSON le:

```
{"Country": "Chile", "City": "Santiago"},
```

by which a client is requesting the current weather conditions in Santiago, Chile. The API would reply with an HTTP response containing the following JSON le:

```
{"timestamp": "14/10/2015 11:59:07",
 "temperature": 25, "Country": "Chile",
 "City": "Santiago", "description": "Sunny"},
```

indicating that the temperature is 25 degrees and the day is sunny. This example illustrates the simplicity and readability of JSON, which partially explains its fast adoption.

With the popularity of JSON it was soon noted that in many scenarios one can bene t from a declarative way of specifying a schema for JSON documents. For instance, in the public API scenario one could use a schema to avoid receiving malformed API calls that may a ect the inner engine of the application. Coming back to the weather application, note that the API calls consist of JSON objects mentioning two strings: a country and a city. What happens if a user does not specify both strings, or if he or she speci es more properties in the JSON object? Similar issues arise when we use a number or a boolean value instead of a string. Without an integrity layer all of these questions need to be taken into consideration when coding the API, and could be avoided if we use a schema de nition to lter out documents that are not of the correct form. A declarative schema speci cation would also give developers a standardised language to specify what types of JSON document are accepted as inputs and outputs by their API.

JSON Schema [20] is a simple schema language that allows users to constrain the structure of JSON documents and provides a framework for verifying the integrity of the requests and their compliance to the API. If we consider again the weather API, by simply adding the following JSON Schema we can assure the correct form of each API call:

```
{
  "type": "object",
  "properties": {
    "Country": {"type": "string"},
    "City": {"type": "string"},
  },
  "required": ["Country", "City"]
  "additionalProperties": false,
}
```

This schema asserts that the received JSON document must be of type object (a collection of key-value pairs), it should contain keys "Country" and "City", both required and with a string value attached to them, and there cannot be any more keys. For example, the JSON le requesting

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2883029.

the weather in Santiago, Chile would comply to this schema, but the JSON file `{"Country": "Croatia", "City": 5}` would not as the value of the city is not a string.

To the best of our knowledge, JSON Schema is the only general attempt to define a schema language for JSON documents, and it is slowly being established as the default schema specification for JSON. The definition is still far from being a standard (the specification is currently in its fourth draft [10]), but there is already a growing body of applications that support JSON schema definitions, and a good deal of tools and packages that enable the validation of documents against JSON Schema. There have been other alternatives for defining schemas for JSON documents, but these are either based on JSON Schema itself or have been designed with a particular set of use cases in mind. To name a few of them, Orderly [16] is an attempt to improve the readability of a subset of JSON Schema, Swagger [2], RAML [29] and Google discovery [12] are proposals for standardising API definition that use JSON Schema, and JSON-LD [26] is a context specific definition to specify RDF as JSON.

Despite all the advantages of a schema definition, the adoption of JSON Schema has been rather slow. One of the issues that have prevented the widespread recognition of JSON Schema as a standard for JSON meta-data is the ambiguity of its specification. The current draft addresses most typical problems that would show up when using JSON Schema, but the definitions lack the detail needed to qualify as a guideline for practical use. As a result we end up having huge differences in the validators that are currently available: most of them work for general cases, but their semantics differ significantly when analysing border cases.

The lack of a formal definition has also discouraged the scientific community to get involved: to the best of our knowledge there has been no formal study of general schema specifications for JSON, nor has there been any formal discussion regarding the design choices taken by the JSON Schema specification. A formal specification would also help the development of automation tools for APIs. There is already software for automatically generating documentation [14] and API clients [13, 15], but all of them suffer from the same problems as validators.

Looking to fill this gap, we present in this paper a formal grammar for the specification of JSON Schema documents, and provide a formal semantics to standardise the meaning of all the features in JSON Schema. For space reasons we cannot present the full formal definition, but we have identified and formalised a semantic core that is enough to express any possible JSON Schema. The full definition can be found on our web page dedicated to JSON Schema [1, 25].

Our framework allows us to conduct a formal study of several aspects of the JSON Schema specification. We begin with the problem of validating a JSON document against a schema, providing tight bounds for the computational complexity of this problem. We also study the expressive power of JSON Schema as a language for defining classes of JSON documents. Since JSON Schema is the only native schema definition for JSON we cannot compare to other standards; instead we provide comparison with respect to automata theory and logic, the two most important theoretical yardsticks for expressive power. These theoretical tools allows us, for example, to conclude that JSON Schema can define relationships that are not available in the schema definitions for XML that are currently used in practice.

We also study the efficiency of JSON Schema in practice using two sets of experiments. First we analyse the impact of validating the most involved features in JSON schema, under JSON documents of increasing size, and conclude that it is not difficult to implement a validation system that scales well with the data. Afterwards we demonstrate a practical use case of increasing importance: a JSON Schema definition for Wikidata [32, 30], the central storage for Wikimedia data [3]. We show the general picture of a Schema for Wikidata, and then validate all 18.4 million entities in its database, at a speed of almost 200 entities per second.

Organisation. In Section 2 we show the problems we run into because of the lack of a formal specification and define syntax and semantics of JSON Schema. In Section 3 we prove the existence of efficient algorithms for the schema validation problem. Next, in Section 4, using our in-house validation tool, we analyse the usability of JSON Schema in practice. We conclude in Section 5.

2. A FORMAL MODEL FOR JSON SCHEMA

One of the main problems of JSON Schema is the lack of a formal specification. To illustrate why this is an issue we created four border-case schemas, and validated them using five different validators. These tests use schemas that are *allowed by the current JSON Schema draft* [10], but the evaluation of their features is not fully specified by the draft. The first test (T1) evaluates whether or not a collection of key-value pairs is considered to be ordered. The second test (T2) checks the behaviour of validators for a schema specifying both that the document is an integer and a string. Next, the test (T3) states that the document is an object, but also adds an integer constraint to it. Lastly, (T4) uses definitions and references to force an infinite loop, while also allowing the object to be a simple string. For space reasons the full code of the tests and their details are left out from the paper, but can be found on our web page [1].

	V1	V2	V3	V4	V5
T1:	N	Y	Y	N	Y
T2:	Y	N	Y	N	Y
T3:	N	Y	N	N	N
T4:	{	{	N	{	{

Y	valid
N	invalid
{	unsupported

Table 1: Validating four documents against four border-case schemas using five different validators. The outcomes stress the difference between the validators' semantics.

Table 1 shows the outcome of this process, It is important to mention that all validators successfully validate the JSON Schema test-suite [4]. As we can see, no two validators behave the same on all inputs, which is clearly not the desired behaviour. This illustrates the need for a formal definition of JSON Schema which will either disallow ambiguous schemas, or formally specify how these should be evaluated.

2.1 JSON Documents and JSON Pointer

We start by fixing some notation regarding JSON documents and introducing JSON Pointer, a simple query language for JSON that is heavily used in the JSON Schema specification. For readability we skip most of the encod-

ing details with respect to these specifications; their formal definition can be found in [5, 18].

JSON Values. The JSON format defines the following types of values. First, `true`, `false` and `null` are JSON values. Any decimal number (e.g. 3.14, 23) is also JSON value, called a *number*. Furthermore, if s is a string of unicode characters then "s" is a JSON value, called a *string value*. Next, if $v_1, \ldots, v_n$ are JSON values and $s_1, \ldots, s_n$ are pairwise distinct string values, then $o = \{s_1 : v_1, \ldots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a key-value pair of o. Finally, if $v_1, \ldots, v_n$ are JSON values then $a = [v_1, \ldots, v_n]$ is a JSON value called an *array*. In this case $v_1, \ldots, v_n$ are called the *elements of* a.

We sometimes use the term JSON document (or just document) to refer to JSON values. The following syntax is normally used to navigate through JSON documents. If J is an object, then J[\key"] is the value of J whose key is the string \key". Likewise, if J is an array, then $J[n]$, for a natural number n, contains the (n-1)-th element of J.

JSON Pointer. JSON Pointers are intended to retrieve values from JSON documents. Formally, a JSON pointer is a string of the form $p = /w_1/\cdots/w_n$, for $w_1, \ldots, w_n$ valid strings using any unicode character.

The *evaluation* Eval(p, J) of a pointer p over a document J is a JSON value that is recursively defined as follows. Assume that $p = /w/p'$. Then Eval(p, J) is:

- the value Eval$(/p', J[n])$, if J is an array, w is the base 10 representation of the number n and J has at least $n + 1$ elements; or
- the value Eval$(/p', J[w])$, if J is an object that has a pair with key "w" (note that we have to put the value of w between quotes to make it a JSON string); or
- the value `null` otherwise.

Example 1. *Consider now an array storing names* `J = [{"name": "Joe"},{"name": "Mike"}]`. *To extract the value of the key* "name" *for the second object in the array, we can use the JSON pointer* `p = /1/name` *which first navigates to the second item of the array (thus obtaining the object* `{"name": "Mike"}`*) and retrieves the value of the key* "name" *from here. Therefore Eval*(p, J) ="Mike".

2.2 Formal Grammar for JSON Schema

JSON Schema can specify any of the six types of valid JSON documents: objects, arrays, strings, numbers, boolean values and null; and for each of these types there are several keywords that help shaping and restricting the set of documents that a schema specifies. As such, in the space given it would be cumbersome to define JSON Schema in its completeness. Instead, we have identified a core fragment that is equivalent to the full JSON Schema specification, and present now its formal grammar and semantics. All of the remaining functionalities in the official JSON Schema draft can be expressed using the functionalities included in this paper. The complete definition can be found in our online appendix [25, 1].

The formal grammar is presented in tables (2-5). It is specified in a visual-based extended Backus-Naur form [28], where all non-terminals are written in bold (and thus everything not in bold is a terminal). Also, for readability, we use string to represent any JSON string, n to represent any positive integer, r to represent any decimal number, Jval to represent any possible JSON document and regExp to represent any regular expression. Note that when these values get instantiated they behave as terminals.

JSDoc	:=	{ (defs ,)? JSch }
defs	:=	"definitions": { string : { JSch } (, string : { JSch })* }
JSch	:=	strSch | numSch | intSch | objSch | arrSch | refSch | not | allOf | anyOf | enum
not	:=	"not": { JSch }
allOf	:=	"allOf": [{ JSch } (, { JSch })*]
anyOf	:=	"anyOf": [{ JSch } (, { JSch })*]
enum	:=	"enum": [Jval (, Jval)*]
refSch	:=	"$ref": "# JPointer"

Table 2: Grammar for JSON Schema Documents

strSch	:=	"type": "string" (, strRes)*
strRes	:=	minLength | maxLength | pattern
minLength	:=	"minLength": n
maxLength	:=	"maxLength": n
pattern	:=	"pattern": "regExp"

Table 3: Grammar for string schemas

Remark. Since every JSON Schema document is also a JSON document, we assume that duplicate keywords cannot occur at the same nesting level.

Overall Structure. Table 2 defines the overall structure of JSON Schema document (JSDoc). It consists of two parts: an optional *definitions* section (defs), that is intended to store other schema definitions to be reused later on, and a mandatory *schema* section (JSch) where the actual schema is specified. In turn, each schema can be either a *string schema* (strSch), a *number schema* (numSch), an *integer schema* (intSch), an *object schema* (objSch), an *array schema* (arrSch), a *reference schema* (refSch), a boolean combination of schemas using not, allOf or anyOf, or simply the *enumeration* of a set of values (enum). Note how reference schemas make use of JSON pointer (JPointer).

Strings. String schemas are formed according to Table 3. We first state that we wish to represent a string using the `"type": "string"` pair, and then we may add additional restrictions to bound the length of the strings or to state that they satisfy a certain regular expression regExp. We illustrate some of these concepts by means of an example.

Example 2. *The following schema* S_1 *specifies strings according to an email pattern. It has no definitions section.*

```
{
    "type": "string",
    "pattern": "[A-z]*@ciws.cl"
}
```

The next schema, S_2*, includes schema* S_1 *as a definition, under the* "email" *key.*

```
{
    "definitions": {
        "email": {
            "type": "string",
            "pattern": "[A-z]*@ciws.cl"
        }
    },
    "not": {"$ref": "#/definitions/email"}
}
```

```
numSch  := "type": "number" (, numRes )*
intSch  := "type": "integer" (, numRes )*
numRes  := min | exMin | max | exMax | mult
min     := "minimum": r
exMin   := "exclusiveMinimum": true
max     := "maximum": r
exMax   := "exclusiveMaximum": true
mult    := "multipleOf": r  (r ≥ 0)
```

Table 4: Grammar for numeric schemas

```
objSch  := "type": "object" (, objRes )*
objRes  := prop | addPr | patPr | req
prop    := "properties": { kSch (, kSch )*}
kSch    := string : { JSch }
addPr   := "additionalProperties": false
req     := "required": [ string (,string)* ]
patPr   := "patternProperties":
                    { patSch (, patSch )*}
patSch  := "regExp" : { JSch }
```

Table 5: Grammar for object schemas

Note that the evaluating the pointer `/definitions/email` *on S_2 yields precisely S_1. Intuitively, this schema is intended to specify all objects that do not conform to S_1.*

Numeric Values. Integer and number schemas have the same structure, shown in Table 4. The pair `"type": "number"` specifies any number, while `"type": "integer"` specifies integers only[1]. We can specify maximum and/or minimum values for numbers and integers (these values are not exclusive unless explicitly stated), and also that numbers and integers should be multiples of another number.

Objects. We specify object schemas with the `"type": "object"` pair, according to the grammar in Table 5. Within objects schemas we may use additional restrictions to control the key-value pairs inside objects. The keyword `required` specifies that a certain string needs to be a key of one of the pairs inside an object, and `properties` is used to state that the value of a key needs itself to satisfy a certain schema. The keyword `patternProperties` works like `properties`, except we bound all key-value pairs whose key satisfies a regular expression, and finally `additionalProperties` controls whether we allow any additional key-value pair not defined in `properties` or `patternProperties`.

Example 3. *Recall the schema from the Introduction describing an API call to the weather app. As the API is expecting a JSON containing a country name and a city name, but nothing else, our schema specifies that these two keys must be present and they have to be of type string. We also use* `required` *and* `additionalProperties` *to specify that the JSON we are sending to the app will contain precisely those two keys and nothing else.*

Arrays. Finally, array schemas are specified with the `"type": "array"` pair, and according to Table 6. There are two ways of specifying what kind of documents we find in arrays. If a single schema follows the `"items"` keyword, then every document in the array needs to satisfy this schema. On the other hand, if an array follows the `"items"` keyword, then it is one-by-one: the i-th document in the specified array needs to satisfy the i-th schema that comes after the `"items"` keyword. We can also set a minimum and/or a maximum number of items, and finally we can use `uniqueItems` to specify that all documents inside an array need to be different.

[1] JSON Schema treats integers as a different type.

```
arrSch  := "type": "array" (, arrRes )*
arrRes  := itemo | itema | minIt | maxIt | unique
itemo   := "items": { JSch}
itema   := "items": [{ JSch} (, {JSch})*]
minIt   := "minItems": n
maxIt   := "maxItems": n
unique  := "uniqueItems": true
```

Table 6: Grammar for array schemas

Example 4. *To illustrate how array schemas work, consider again the API described in the Introduction. Imagine now that our API also allows us to ask information about the weather for several places simultaneously. An obvious way to model such requests is by using JSON arrays, where each item of the array is a single call as in Example 3. To check that the requests we send are using the correct format we could validate them against the following schema (The reference is assumed to return the schema of Example 3):*

```
{
  "type": "array",
  "items": {"$ref": "#/definitions/basic_call"}
}
```

2.3 Semantics

The idea is that a JSON document satisfies a schema if it satisfies all the keywords of this schema. Formally, given a schema S and a document J, we write $J \models S$ to denote that J satisfies S. We separately define $\models$ for string, number, integer, object and array schemas, as well as for their combinations or enumerations.

Combinations and References. Let S be a boolean combination of schemas, an enumeration or a reference schema. We say that $J \models S$, if one of the following holds.

- S is `"enum":`$[J_1, \ldots, J_m]$ and $J = J_\ell$, for some $1 \leq \ell \leq m$.
- S is `"allOf":`$[S_1, \ldots, S_m]$ and $J \models S_\ell$, for all $1 \leq \ell \leq m$.
- S is `"anyOf":`$[S_1, \ldots, S_m]$ and $J \models S_\ell$, for some $1 \leq \ell \leq m$.
- S is `"not":`S' and $J \not\models S'$.
- S is `"$ref":"#`p`"` for a JSON pointer p; Eval(p, D) is a schema and $J \models$ Eval(p, D), with D the JSON document containing S.

Note that if Eval(p, D) returns `null` then `"$ref":"#`p`"` is not satisfiable, and likewise if Eval(p, D) returns a JSON value that is not a schema.

Strings. Let S be a string schema. Then $J \models S$ if J is a string, and for each key-value pair p in S that is not `"type": "string"` one of the following holds:

- p is `"minLength":n` and J is a string with at least n characters.

- p is "maxLength":n and J is a string with at most n characters.
- p is "pattern":e and J is a string that belongs to the language of the expression e.

Example 5. *Consider again schemas S_1 and S_2 from Example 2. Furthermore, let J be \admin@ciws.cl". We then have that $J \models S_1$, because J is a string that belongs to the regular expression in S_1. On the other hand, since the pointer* /definitions/email *retrieves once again S_1, schema S_2 is actually equivalent to*

```
{
    "not": {
        "type": "string",
        "pattern": "[A-z]*@ciws.cl"
    }
}
```

and thus $J \not\models S_2$.

Numeric Values. Let S be a number (respectively, integer) schema. Then $J \models S$ if J is a number (resp. integer), and for each key-value pair p in S whose key is not "type", "exclusiveMinimum" or "exclusiveMaximum" one of the following holds:

- p is "minimum":r and J is strictly greater than r.
- p is "minimum":r, J is equal to r and the pair "exclusiveMinimum": "true" is not in S.
- p is "maximum":r and J is strictly lower than r
- p is "maximum":r, J is equal to r and S the pair "exclusiveMaximum": "true" is not in S.
- p is "multipleOf":r and J is a multiple of r.

Objects. Let S be an object schema. Then $J \models S$ if J is an object, and for each key-value pair p in S that is not "type": "object" one of the following holds:

- p is "properties":$\{k_1 : S_1, \ldots, k_m : S_m\}$ and for every key-value pair $k : v$ in J such that $k = k_j$ for some $1 \leq j \leq m$ we have that $v \models S_j$.
- p is "patternProperties":$\{$"e_1": $S_1, \ldots,$ "e_m": $S_m\}$ and for every key-value pair $k : v$ in J and every e_j, with $1 \leq j \leq m$, such that k is in the language of e_j we have that $v \models S_j$.

  Remark. If the keyword matches more than one pattern property then it has to satisfy all the schemas involved.
- p is "required":$[k_1, \ldots, k_m]$ and for each $1 \leq j \leq m$ we have that J has a pair of the form $k_j : v$.
- p is "additionalProperties": false and for each pair $k : v$ in J, either S contains "properties":$\{k_1 : S_1, \ldots, k_m : S_m\}$ and $k = k_j$ for some $1 \leq j \leq m$, or S contains "patternProperties":$\{$"e_1": $S_1, \ldots,$ "e_m": $S_m\}$ and k belongs to the language of e_j, for some $1 \leq j \leq m$.

Arrays. Let S be an array schema. Then $J \models S$ if J is an array, and for each key-value pair p in S that is not "type": "array" one of the following holds:

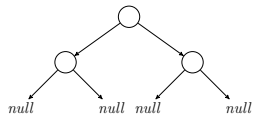
- p is "items":$\{S'\}$ and for each item $J' \in J$ we have that $J' \models S'$.
- p is "items":$[S_1, \ldots, S_m]$, $J = [J_1, \ldots, J_\ell]$ and $J_i \models S_i$ for each $1 \leq i \leq \min(m, \ell)$.
- p is "minItems":n and J has at least n items.
- p is "maxItems":n and J has at most n items.
- P is "uniqueItems": true and all of J's items are pairwise distinct.

Example 6. *As our final example consider the schema*

```
{
 "definitions": {
   "S": {
     "anyOf": [
       {"enum": [null]},
       {"allOf": [
          {"type": "array",
           "minItems": 2,
           "maxItems": 2,
           "items": [
             {"$ref": "#/definitions/S"},
             {"$ref": "#/definitions/S"}]
          },
          {"not": {"type": "array", "uniqueItems": true}}
       ]}
   ]},
   "$ref": "#/definitions/S"
 }
}
```

This schema defines a nesting of arrays where each element is itself a json document that conforms to S*: either the* null *JSON or an array with exactly two elements that again must conform to* S*. We can look at these objects as binary trees, where arrays represent inner nodes and* null *JSONs represent leafs. For example, here is a tree representation of a JSON document that satisfies to this schema.*

```
[
    [null, null],
    [null, null]
]
```



It is straightforward to show that any document validating against this schema has to be a description of a complete binary tree. The latter constraint is enforced by having the "not": {"type": "array", "uniqueItems": true} *clause in the object description, thus guaranteeing that if a node has a child, it has to have precisely two children that are equal to each other.*

Well Formedness. The formal grammar still allows for problematic schemas, such as the following.

```
{
    "definitions": {
        "S": {"not": {"$ref": "#/definitions/S"}}
     },
     "$ref": "#/definitions/S"
}
```

The above de nes a schema that is both S and the negation of S, and is therefore ill-designed. What is worse, the majority of the validators we tested run into an in nite loop when trying to resolve the references of this schema. In fact, one of the tests in Table 1 used a schema similar to this one.

To avoid these problematic cases we introduce the notion of a *well-formed* schema. Formally, let S be a JSON schema document and S_1

corresponds to a JSON Pointer that retrieves S_j. For instance, the graph of the document above has only the node S and the only edge is a self loop on S. Edges are only added if S_i is a combination of schemas, not if for example S_i is an object schema and the reference for S_j is under a "properties" keyword.

We then say that S is a well formed schema if such a graph is acyclic. For the rest of the paper we consider only well formed schemas, and we propose to add this condition to the standard as well. Note also that well formedness can be checked in linear time.

Remark. We stress again that our de nition does not cover all of the syntactic nuances of the JSON Schema speci cation [20]. What it can do, however, is to provide a reformulation of any JSON Schema document as speci ed in [20] in a concise syntax and such that the original schema and its reformulation de ne the same set of JSON documents.

3. FORMAL ANALYSIS

In this section we show several results regarding the efficiency of working with JSON Schema and its expressive power. We begin with the computational cost of checking if a document conforms to a schema. We then compare JSON Schema to several well established theoretical formalisms such as nondeterministic nite state automata, tree walking automata and monadic second order logic.

3.1 Validation

The most important problem related to JSON schema is to determine if a JSON document J conforms to a schema S. This problem is called *JSON Schema validation* and is formally de ned as follows.

Problem:	JSchValidation(J, S)
Input:	JSON document J and schema S.
Question:	Does $J \models S$?

Since most of JSON is used to transfer data between web applications, developing e cient algorithms for JSON Schema validation is of critical importance. It is thus important to understand the computational complexity of the schema validation problem, as this gives us a good starting point for the design of e cient validation algorithms.

We show that the problem is always in PTIME, and can be solved in linear time with respect to both the schema and the document as long as the schema does not use the uniqueItems keyword. However, the problem remains PTIME-hard, even for schemas using a very limited set of keywords. This illustrates that although it is possible to solve the validation problem e ciently, it is still harder than for example checking if two nodes in a graph are connected by a path, or determining whether a word belongs to the language speci ed by a regular expression.

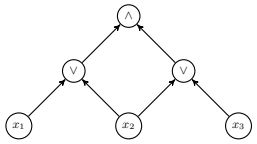
Let us begin with the PTIME upper bound. Given a schema S and a JSON document J, we derive a simple algorithm that runs in time $O(|J| \cdot |S|)$ as long as the schema does not contain the uniqueItems keyword. The algorithm works as follows: we process the document restriction by restriction, while checking conformance to the corresponding subschema in S. The running time is linear because correspondence to each keyword in JSON Schema can be checked in linear time (except for uniqueItems). If S does contain uniqueItems, then we may now need to check whether all the elements of a given array J are unique. This check can be performed in time $O(|J| \cdot \log |J|)$ by rst sorting the array J, thus raising the total bound to $O(|J| \cdot log|J| \cdot |S|)$. One can in fact show that this bound is tight, as it is equivalent to computing the lower bound of any comparison based sorting algorithm [7]. We do note that the $O(|J| \cdot |S|)$ bound should remain in most practical scenarios even in the presence of uniqueItems, since the uniqueness is likely to be performed using a hash table. Regardless, we obtain the following:

Proposition 7. *The problem* JSchValidation(J, S) *is in* PTIME.

And as promised, the matching lower bound is obtained using very simple schemas.

Proposition 8. *The problem* JSchValidation(J, S) *is* PTIME*-hard even if S contains just the restrictions* allOf, anyOf *and* enum.

Proof Sketch. The proof is by reduction from the Monotone Circuit Value problem, which is known to be PTIME-complete [11]. Given a circuit C and a valuation τ for the gates of C, we provide a LOGSPACE reduction that traverses the circuit in a depth- rst fashion, starting from the root. If we encounter an AND gate, we add an allOf restriction to the schema, with a number of subschemas equal to the number of inputs the gate has. If the gate is an OR we add an anyOf restriction. In both cases we repeat the process until we reach the leafs of C. At this point, we force the JSON instance to have either the value true or false, depending on τ, by using the enum restriction. Our document J is simply the value true. We illustrate how the reduction works in Figure 1.




```
{"allOf":[
   {"anyOf":[
      {"enum":[false]},
      {"enum":[false]}]},
   {"anyOf":[
      {"enum":[false]},
      {"enum":[true]}]}
]}
```

Figure 1: Schema for the circuit C with input values $\tau(x_1)$ = $\tau(x_2)$ = false and $\tau(x_3)$ = true.

In Figure 1 the allOf construct corresponds to the AND gate of the circuit, while the two anyOf subschemas simulate the OR gates. The input values are coded using enum in order to equal constants true and false.

It is easy to see that $J \vDash S \iff \tau(C)$ = true. □

3.2 Expressive Power

So far we have seen many examples of how JSON Schema can be used in practice, but we still do not know much about the classes of JSON documents that the JSON Schema speci cation can express, and which ones it cannot. To answer these questions it is convenient to study whether JSON Schema can simulate any of the well established formalisms for de ning languages. We provide two such comparisons: with respect to automata and with respect to logic.

Automata. Most schema de nitions for other semistructured data paradigms are heavily based on automata. In the case of XML, for example, there has been a lot of study in linking schema de nitions to di erent versions of tree automata (c.f. [23]). It is therefore useful to compare with

automata formalisms, if only to understand how much does JSON Schema depart from XML schema formalisms.

We begin by showing that JSON schema can de ne any standard non-deterministic nite automaton (NFA). Obviously we can do this since we have the `pattern` keyword to de ne strings that satisfy any regular expression. However, we show that even if we are left with just a few keywords (and no `pattern`) we can still simulate automata, with the help of a very simple coding. This shows that JSON Schema is inherently as expressive as NFAs, and we later use these results to argue that the power of JSON Schema is, in essence, at least comparable to most XML schema speci cations.

To formally state this result consider the coding of a word w over a nite alphabet into a JSON document $J(w)$, that treats each letter as a property and the following letters as its subproperties until it reaches the end of the word, which is represented by a `null` value. For example, the word abc is coded as a JSON document `{"a": {"b": {"c": null}}}`. Similarly, ad is coded as `{"a": {"d": null}}`.

The idea is to show that for every NFA A one can construct a schema S_A such that a word w belongs to the language of A if and only if $J(w) \models S_A$. To illustrate this claim consider the automaton $\mathcal{A}$ in Figure 2.

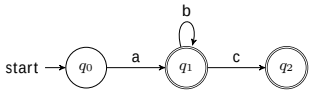


Figure 2: The automaton $\mathcal{A}$ recognising `ab* | ab*c`.

To obtain a schema that will accept (up to our coding) only the words in the language of the automaton from Figure 2 we proceed as follows. First, in the `"definitions"` section of our schema we de ne each state of the automaton. Figure 3 illustrates how is this done for the automaton above. Namely, we have a schema for q_0, q_1 and q_2. Each of these schemas is intended to code the transition from the state it describes. This is achieved by declaring that each state is an object whose properties code the transitions leaving the state. For instance, in order to simulate that we can move from the state q_0 to the state q_1 reading the letter a, we add `"a": {"$ref": "#/definitions/q1"}` to the properties of the schema for q_0. Note that here we use `$ref` to switch to the schema of q_1 and follow the transition. Likewise, to re ect that a non deterministic choice is available, we use the `anyOf` keyword. For instance, this is re ected in Figure 3 when describing the transitions of q_1. Finally, in order to signal that a sate is accepting, we allow it to be of type `null`, such as e.g. for q_1 and q_2. The full transformation is given in Figure 3.

It is now straightforward to see that a word w belongs to the language of the automaton $\mathcal{A}$ from Figure 2 if and only if $J(w)$ validates against the schema from Figure 3. Note that the documents conforming to the schema above are not allowed to have additional properties, so the document J(ad) will not validate against the schema. On the other hand, J(abc) does validate as desired.

Although the procedure described above treats one particular automaton, it also shows how to construct a schema for an arbitrary automaton. We therefore obtain the following.

Proposition 9. *JSON Schema can simulate finite state automata even when it only uses definitions, references, single enumeration and combinations of schemas.*

```
{
   "definitions": {
      "q0": {
         "type":"object",
         "properties": {
            "a": {"$ref": "#/definitions/q1"}
            },
          "additionalProperties": false},
      "q1": {
         "anyOf": [
            {"type":"null"},
               "ab: {"$ref": "#/definitions/q1"}
```

what we call simpli ed schemas, where only object schemas and strings schemas are allowed (but not integer, number or array schemas). However, one can show that our results continue to hold for any fragment of the full JSON Schema speci cation [1, 25, 20] that does not use the `uniqueItems` keyword (which we show not to be de nable in MSO).

To understand the connection with logic it is best to consider every JSON instance J as an unranked unordered tree $T(J)$ (recall that we do not consider arrays here), whose leafs are either empty object instances or strings. For example, Figure 4 shows a simple JSON document and its representation as a tree structure.

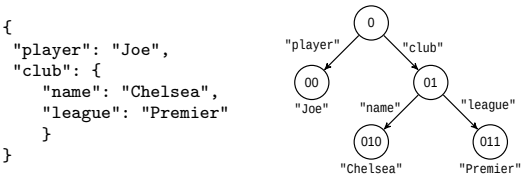


Figure 4: A JSON J and its tree representation $T(J)$.

We then represent these trees as MSO structures using binary relations $Child$, Key and $Value$, and an unary relation $Root$. The interpretation of $Child$ and $Root$ is the usual one; additionally we use Key to store the key of the key-value pairs in the document (such as `"player"` or `"name"` in Figure 4), and $Value$ to store the value of a given string node (such as `"Joe"` or `"Chelsea"`).

The key observation is that each simpli ed schema can be described using an MSO formula over $T(J)$. Instead of giving the full translation we illustrate how it works using the following example. Consider rst the schema S below.

```
{
 "type": "object",
 "properties": {"player": {"type": "string"}},
 "required": "player"
}
```

This schema speci es all objects that have a player attribute whose value is a string. In particular, the document from Figure 4 validates against this schema. An MSO formula equivalent to this schema would be:

$$\exists x \exists y \Big(Root(x) \wedge Child(x,y) \wedge Key(\texttt{"player"}, y) \wedge \exists z \big(Value(y,z) \big) \Big)$$

Intuitively, this formula checks the existence of a child of the root that is accessible via an edge labelled `"player"` whose value is a string. Using the codi cation from Figure 4 this is equivalent to saying that the JSON document has a key-value pair with the key being `"player"` and the value a string, as desired. Other JSON Schema constructs can be simulated by MSO operators in a similar way. Note that we need to use second order properties only to deal with de nitions and references, and to simulate regular expressions (here we use the $Value$ relation in a non trivial way); all other keywords are expressible in rst order logic. Generalising this construction in a similar way as done in [22], we obtain the following.

Theorem 10. *For any simplified schema S there exists an MSO formula F_S such that for every JSON document J we have that $J \models S$ if and only if $T(J) \models F_S$.*

Observe now that, since simpli ed schemas can be described using MSO formulas, and since for each JSON document J the structure $T(J)$ has bounded tree width (as it is essentially a tree), Courcelle's theorem [8] applies. This once again gives us that the validation problem can be solved in time that is linear in the size of the input JSON document.

As we have mentioned, one can extend this result to apply for every JSON schema not using the `uniqueItems` keyword. On the other hand, for the case of schemas using `uniqueItems` we can also show that these schemas are not de nable in MSO. To see this, recall Example 6, that expresses documents representing complete binary trees. It is not di cult to show that this property cannot be accepted by a non-deterministic tree automata, and thus it cannot be expressed in MSO, as tree automata and MSO are equivalent in expressive power [6]. This also explains why `uniqueItems` keyword cannot be validated in time that is linear in the size of the input document, as discussed previously.

4. PRACTICAL CONSIDERATIONS

In this section we conduct an experimental analysis on the e ciency and applicability of JSON schema. We rst run a series of experiments in which we test our own validator under JSON documents of increasing sizes; the results are summarized in Subsection 4.1. We then show in Subsection 4.2 a real use case where JSON Schema can be naturally applied in practice, namely the Wikidata database [32].

4.1 Experimental Analysis

To evaluate how the validating process fares when facing more involved features of JSON schema we conducted four sets of experiments covering a wide range of features: recursive calls using references, objects with a high number of nesting or additional properties, and the validation of `uniqueItems` for arrays with a big number of items.

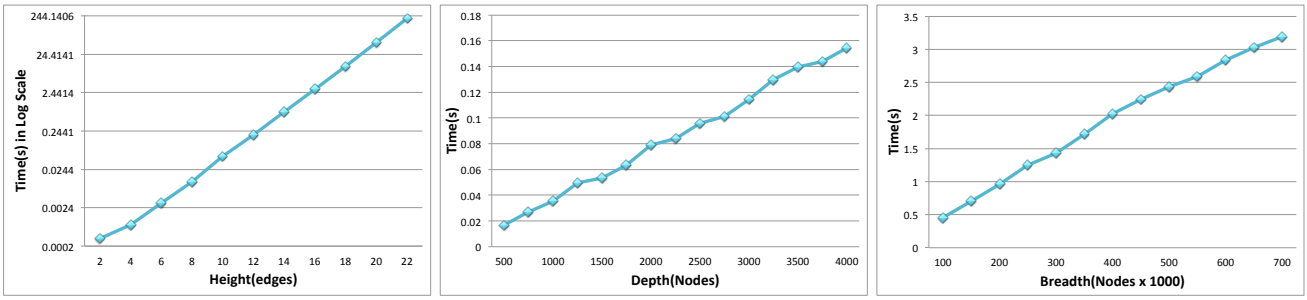
We implemented our own validator that works exactly as described in the formalisation of this paper. For each experiment we created a schema, and measured the time that it takes to validate documents of increasing size against this schema. We wanted to see how our validator performs on a typical personal computer, so we used a machine with 8 GB of RAM and a 2.9 GHz Intel Core i5 processor.

We begin by testing how our validator performs when dealing with recursive calls that use JSON references. The rst schema we test against checks if our JSON document is a binary tree, so we validate binary trees of increasing depth[2]. The results of this experiment are displayed in Figure 5 (a). Observe that this gure uses a logarithmic scale for time, which is only fair, since the number of nodes in the tree increases exponentially with the depth (for instance, for depth 20 we have around 2 million nodes, while for depth 22 this raises to over 8 million).

The second schema we test against uses `$ref` in order to reach a basic type in a nested JSON object. The documents we are validating against this schema will be objects with increasing nesting depth (i.e. objects of the type `{"x":{"x":{"x": true}}}`) and the results are presented in Figure 5 (b). As expected from the results of Section 3.1 the validator displays linear behaviour.

The third schema simply tests that each property of an object is of a certain type, against JSON objects that contain

[2]Test data can be found at [1].



(a) Binary tree check (b) Finding the depth (c) Trees with multiple children

Figure 5: Performance of di erent tests on documents of increasing size.

an increasing number of key-value pairs. As we can see from Figure 5 (c), the validation is very e cient, even when we are dealing with hundreds of thousands of key-value pairs.

The nal schema tests if an array has unique items, validating arrays that contain a varying number of complete binary trees with around four thousand nodes. The performance, shown in Figure 6, con rms the results from Section 3.1 which state that we can not expect linear time validation when dealing with schemas using the `uniqueItems` keyword. The running time however is quite reasonable, even when dealing with a large quantity of complicated objects.

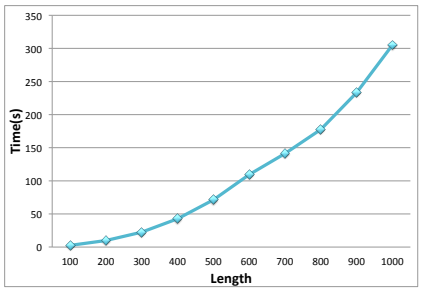


Figure 6: Testing `uniqueItems` for arrays of varying size.

It is important to note that these experiments use very demanding documents (e.g. complete binary trees). This gives us realistic upper bounds on the performance, but running times in practice should be much faster.

As we can see the results we obtain are encouraging and suggest that even when les weighing up to a hundred MB are validated against non trivial schemas we can get the answer in at most a few minutes using a typical developer machine. This additionally illustrates that JSON Schema validation is a process that does not require server side support and can thus be done locally when working with the JSON data format. Therefore our experimental results imply that including JSON Schema validation in the application development process is already a viable option.

4.2 Use Case: Wikidata

JSON is also used for storing data, and is the format of choice for several document-oriented databases such as MongoDB [24] and CouchDB [27]. The advantages of having a schema de nition in a classical database scenario have long been recognised by the community, and we believe that JSON databases would bene t from JSON Schema in the same way. To analyse these advantages we choose to study Wikidata and show how JSON Schema can be used to describe the structure of its documents.

Wikidata is a free linked database that acts as central storage for the data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wikisource, and others [32]. As far as we know, Wikidata has the biggest JSON database on the web with more than 60GB of information. The data is constantly made available in the form of JSON data dumps, and there is also a wide range of APIs and tools available for connecting to Wikidata. As of today, consuming this data is a complicated process, as users need to learn the details of the data format by trial and error, and small changes in the underlying structure of the les usually 2uref JSat

```
"claims": {
        "P610": [
          {...
                "value": {
                  "entity-type": "item",
                  "numeric-id": 43105
                }, ...
          } ...
```

This object tells us that entity Q46 is connected with the item Q43105 (\Mount Elbrus") through the property P610 (\Highest Point"), or that the highest point of Europe is Mount Elbrus. Each key-value pair in `claims` represents a particular statement about the entity (or an array of statements using the same property). These pairs have the property as the key, and the value of these pairs contains the rest of the statement (among other things).

Schema. As we have mentioned, the outermost structure is an object with at least four key-value pairs. The keys are id, type, labels and claims. The value of id is a string that starts with P or Q, the type can be either item or property, and labels and claims are more complex objects; we de ne their schemas under the de nitions section and then reference it using the `$ref` keyword. The following schema describes the class of objects we have discussed.

```
{
  "definitions": {
    "labels": {...},
    "claims": {...}
  },
  "type": "object",
  "required": ["type","id","labels","claims"],
  "additionalProperties": false,
  "properties": {
  "id": {
      "type": "string",
      "pattern": "^(P|Q)[0-9]+$"
    },
    "type": { "enum": ["item","property"] },
    "labels": {"$ref": "#/definitions/labels"},
    "claims": {"$ref": "#/definitions/claims"}
  }
}
```

The value of labels is captured by the following schema:

```
"labels": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "required": ["language","value"],
        "properties": {
          "language": {"type": "string"},
          "value": {"type": "string"}
        }
      }
    }
```

Note the nesting under the `additionalProperties` keyword. This states that the outermost object may have any number of pairs, but all of them need to satisfy the schema stated under the additional properties keyword.

The incomplete schema below conveys the most important information regarding claims: each pair in claims needs to have a key that starts with P (i.e. a property), and the value of each of these statements lies further inside the document.

```
"claims": {
      "type": "object",
      "patternProperties": {
        "^P[0-9]+$": {
          ...
          "entity-type": {"type": "string"},
          "numeric-id": {"type": "integer"}
          ...
        }
```

Note that each of the keys mentioned under `claims` is an entity on its own, and therefore it has its own document. For instance, we can nd the document for property P610 on the Wikidata database, with further statements about this property (such as the statement \see also": \deepest point"). It is important to keep these references up to date, as it is easy to loose these links when data is deleted or modi ed. Unfortunately JSON Schema is not designed to enforce this type of constraints. We believe this is an important direction for future work on JSON Schema.

Validation. Wikidata publishes a regular dump of their data [31]. This dump is a JSON array, and each of the items in this array is a di erent Wikidata entity.

To show the practical applicability of JSON Schema we decided to run a complete validation of Wikidata's more than 18 million entities. To treat each entity as a separate JSON document we created a simple script that extracts each entity of the document and validates it against our schema. We used a computer with 4 GB of RAM and a Quad-Core Intel Xeon E5 3.7 GHz processor. The results of our validation are given in the table below.

Entities validated:	18375981
Total time:	27.251 hours
Entities per second	187.312
Average time per entity:	0.005 seconds

The size of the database is almost 60 gigabytes, so any sequential algorithm is going to need some time to process this data. However, the total running time can easily be shortened by running several validations in parallel; what is important for us is that validating a single JSON document in practice takes just a few milliseconds, and our validator can handle nearly 200 documents per second. We believe this is a strong indicator of the feasibility of implementing

6. REFERENCES

[1] Online Appendix. http://web.ing.puc.cl/~jreutter/JSch, 2015.

[2] Swagger: The World's Most Popular Framework for APIs. http://swagger.io/, 2015.

[3] Wikimedia. https://www.wikimedia.org/, 2015.

[4] J. Berman. JSON Schema Test Suite. https://github.com/json-schema/JSON-Schema-Test-Suite, 2015.

[5] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12 { 75, 1990.

[9] ECMA. The JSON Data Interchange Format . http://www.ecma-international.org/publications/standards/Ecma-404.htm, 2013.

[10] F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. http://json-schema.org/latest/json-schema-core.html, 2013.

[11] L. M. Goldschlager. The Monotone and Planar Circuit Value Problems Are Log Space Complete for P. *SIGACT News*, 9(2):25{29, July 1977.

[12] Google. Google API Discovery Service. https://developers.google.com/discovery/, 2015.

[13] I. G. group. Heroics: Ruby HTTP client for APIs represented with JSON schema. https://github.com/interagent/heroics, 2013.

[14] I. G. group. Prmd: JSON Schema tools and documentation generation for HTTP APIs. https://github.com/interagent/prmd, 2013.

[15] I. G. group. Schematics: A Go point of view on JSON Schema. https://github.com/interagent/schematic, 2013.

[16] L. Hilaiel. Orderly. http://orderly-json.org/, 2015.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003

[18] Internet Engineering Task Force (IETF). JavaScript Object Notation (JSON) Pointer. https://tools.ietf.org/html/rfc6901, April 2013.

[19] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc7159, March 2014.

[20] json-schema.org: The home of json schema. http://json-schema.org/.

[21] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[22] L. Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[23] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of xml schema. *ACM Transactions on Database Systems (TODS)*, 31(3):770{813, 2006.

[24] MongoDB Inc. The MongoDB3.0 Manual. https://docs.mongodb.org/manual/, 2015.

[25] J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoc. JSON Schema: syntax and semantics. http://cswr.github.io/JsonSchema/, 2015.

[26] M. Sporny, G. Kellogg, and M. Lanthaler. JSON-LD 1.0: A JSON-based Serialization for Linked Data. http://www.w3.org/TR/json-ld/, January 2014.

[27] The Apache Software Foundation. Apache CouchDB. http://couchdb.apache.org/, 2015.

[28] The International Organization for Standardization (ISO). ISO/IEC 14977:1996 - Extended BNF. http://www.iso.org/iso/catalogue_detail?csnumber=26153, 1996.

[29] The RAML Workgroup. RAML: RESTful API Modeling Language. http://raml.org/, 2015.

[30] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78{85, 2014.

[31] Wikidata. Wikidata:Database download. https://www.wikidata.org/wiki/Wikidata:Database_download, 2015.

[32] Wikimedia. Wikidata: The Free Knowledge Base. http://www.wikidata.org, October 2015.