JSON Model: a Lightweight Featureful Description Language for JSON Data Structures

Fabien Coelho, Claire Yannou-Medrala

JSON Model: a Lightweight Featureful Description Language for JSON Data Structures

Fabien Coelho and Claire Yannou-Medrala
firstname.lastname@minesparis.psl.eu
Centre de recherche en informatique, Mines Paris – PSL University
France

ABSTRACT

JSON is a simple *de facto* standard cross-language textual format used to represent, exchange and store structured data in computer systems. Data schemas need to be described for documentation and verification purposes. Three JSON-based schema description languages have been proposed for JSON so far: JSON Schema, JSound and JSound-C. These languages are quite verbose and have a lax validation semantics: 60% of public schemas have been found defective [15] because JSON Schema is particularly error-prone. We introduce JSON Model, a work-in-progress alternative to previous proposals which is both lightweight and featureful, discuss key design choices and possible variants.

KEYWORDS

JSON Model; Schema description language; DSL;

1 INTRODUCTION

The JSON [8] *JavaScript Object Notation* format has become in recent years an ubiquitous cross-language *de facto* standard to represent, exchange and store data between computer applications, partially replacing XML [12]. Its success can be attributed to the extensive use of JavaScript in web and mobile development. Like the more verbose XML, JSON can be parsed without knowing in advance the expected structure. It allows to serialize in textual form simple data structures (Figure 1) built upon the *null* value, booleans, numbers, Unicode strings, arrays (aka list, tuple, sequence, set) and objects (aka struct, record, dict, map, association, key-value pairs). Its drawbacks include the limited number of types, the absence of a syntax for comments, its unbounded numbers which cannot express some values (e.g. *NaN*), the restriction of object properties (aka key, attribute, field) to strings, and that only tree structures can be serialized, i.e. there is no sharing of values or cycle. Thanks to these simple features, a wide range of libraries and tools are available for many programming languages and systems beyond JavaScript [10] including Python, Java, Shell, and SQL.

```
{
  "name": "Susie",
  "age": 6,
  "friends": [ "Calvin", "Hobbes" ]
}
```

Figure 1: A JSON object with 3 properties

JSON data are mostly generated and processed automatically from a programming language: Humans tend to prefer unquoted structured languages such as YAML to write structured files (e.g. configurations), or lightweight markup languages such as Markdown for text formatting. JSON has two main overlapping use cases:

API Data for simple structures exchanged between computer systems at API interfaces, for instance between web front-ends and back-ends in multitier architectures;

Documents for possibly large loosely-structured textual data which are stored, transmitted, processed and finally displayed for direct human consumption.

The overlap comes from development practices based on JavaScript dynamic typing (lack of) discipline and loose document-oriented schema-less databases such as MongoDB which do not require data schemas to be formally declared. Another example of overlapping usage is open data documentations: They are structured data often accessed through an API, but they also need extensive meta-data for documentation purpose.

In this paper, we present the design of JSON Model, a schema description language for JSON data, which emphasizes compactness and expressiveness, with a particular interest in the API data-structure use case. Section 2 first discusses existing data-structure description languages, with a focus on JSON. Section 3 presents JSON Model design choices, syntax and semantics. Section 4 discusses particular aspects and possible alternatives, before concluding in Section 5.

2 RELATED WORK

As data structures are a key component of programming languages, describing them with various degrees of constraints is typically included in language syntaxes. This also applies to dynamically typed languages, e.g. TypeScript [5] has been developed to allow type declarations with JavaScript [10].

```
TABLE Person(
  name TEXT,
  age INT,
  friends TEXT[]
);
```

(a) SQL

```
person =
  name:string x
  age:int x
  friends:string*
;
```

(b) Newgen

```
struct Person {
  string name;
  short age;
  sequence<string>
    friends;
};
```

(c) IDL

Figure 2: Descriptions for a Person

When considering cross-language features such as data interchange, language-independent description languages can been used. Three examples are shown in Figure 2. Example (a): SQL relational databases are neutral from programming languages which can access data through libraries. Example (b): Newgen [11] provides a data description language, compiler and libraries which allows to share data between C and Common Lisp. Example (c): CORBA [6] includes an interface description language (IDL) to describe structures which can be exchanged between clients and servers These

description languages need specialized tools to manipulate actual data, and require to learn a syntax.

Another approach is to use a file format such as XML and JSON which can be parsed without knowing in advance its structure. For these, data description can be done later, e.g. with a DTD or XSD for XML, and with JSON Schema, JSound or JSound-C for JSON.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "A Person",
  "type": "object",
  "properties": {
    "name": { "type": "string", "minLength": 1 },
    "age": { "type": "integer", "minimum": 0 },
    "friends": {
      "title": "The Person's Friends",
      "type": "array",
      "items": { "type": "string", "minLength": 1 },
      "minItems": 1
  } },
  "required": [ "name", "age" ],
  "additionalProperties": false
}
```

Figure 3: Tight JSON Schema for Figure 1

JSON Schema [14] has been under development as an Internet Draft for over 13 years and 10 versions. The latest proposal (2020-12) defines 60 keywords to describe JSON structures, much like classic programming language data structures, but also includes constraints such as element sizes (strings, arrays…), specific properties (uniqueness), regular expressions to validate unexpected property names, schema composition (`allOf anyOf oneOf`) and logical assertions (`if then else not`). It has an open-document mindset: schemas are loose by default, allowing any type or property unless explicitly stated otherwise. As JSON Schema uses JSON to describe JSON, there is a meta-schema which can validate itself. Figure 3 shows a tight schema suitable for Figure 1 data, which is 4 times larger than the sample it describes. The overall complexity [2, 4, 15] results in a majority of public schemas to be defective because of mistyping, misplacement, misnaming and other mistakes.

JSound [1] is an alternative JSON-based description format that has been proposed to describe JSON and TYSON (a typed extension of JSON) data structures (Figure 4). It is inspired by XML Schema and simpler than JSON Schema. It differs from the later in its syntax and addresses some of its shortcomings, e.g. types are mandatory, and extensive types appropriate to programming are available. Some JSON Schema features, such as composition operators or constraining unknown property names, are not available, but it provides inheritance. It shares the same open-document mindset: Properties are optional by default, and unspecified properties must be explicitly forbidden. JSound comes with an interesting compact variant called JSound-C (Figure 5). This variant is incompatible with full JSound: It cannot be mixed with it. The whole description syntax must be switched if any feature is not available. In particular, the compact version does not allow tight object definitions.

In this paper, we propose an alternate JSON system for describing JSON data which is even more compact than JSound-C, but still retains most (useful) features from JSON Schema.

```
{
  "types": [
    {
      "name": "Person",
      "kind": "object",
      "content": [
        { "name": "name", "type": "string", "required": true },
        { "name": "age", "type": "integer", "required": true },
        {
          "name": "friends",
          "type": {
            "name": "list-of-friends",
            "kind": "array",
            "content": "string"
        } }
      ],
      "closed": true
} ] }
```

Figure 4: Tight JSound for Figure 1

```
{
  "Person": {
    "!name": "string",
    "!age": "integer",
    "friends": [ "string" ]
} }
```

Figure 5: Loose JSound-C for Figure 1

3 JSON MODEL

This section introduces the design criteria, the basic syntax and semantics, the representation of values (constants, references, regex), constraints, composition and complements.

3.1 Design Criteria

JSON data structures exchanged with REST API by web and mobile applications are typically not-deeply-nested objects with few properties. Data structures are usually tight, because from a programming perspective it is easier to write code when assumptions can be made about actual types. For this kind of use cases, we want to provide simple and compact type declarations that are smaller than corresponding values. Furthermore, we still want to make it possible to constrain values with regular expressions and sizes. We thus define the following wish list, which sets desirable characteristics for a new JSON meta-data description language.

self-hosted description in JSON, with a tight meta schema.
compact much shorter than JSON Schema and JSound.
convenient optimized for the simple data structure API use case.
intelligible easy to understand for humans.
frugal avoid multiple redundant concepts.
tight models should be tight by default.
expressive power of description comparable to JSON Schema.
composable by defining and referencing new types.
efficient allow fast (compiled) validation functions.
extensible it should possible to add new concepts if needed.

3.2 Model Basics

In order to have a compact and intelligible representation, JSON Model relies on type inference [9], so that each type is represented

by its simplest value. The 5 basic types and 2 constructs of JSON are thus expressed directly: `null` as a model stands for the `null` value, `true` is a boolean value, i.e. `true` or `false`, `0` is an integer, `0.0` is a number, `""` (empty string) is a string, `[]` and `{}` are *empty* arrays and objects. Obviously, these two later constructs are not very useful in themselves.

Arrays are typically used to hold extended homogeneous lists or fixed tuples of heterogeneous items, so these two usage patterns are prioritized. An extended list is expressed as an array of **one** type: `[ true ]` is a list of booleans. A tuple is expressed as an array of two or more types: `[ "", 0, [ 0.0 ] ]` is thus a tuple composed of a string, an integer, and a list of numbers. Although this approach creates a non uniformity as a tuple of one element cannot be simply expressed, this particular degenerate case does not seem worth preserving, especially as the homogeneous list use case is quite frequent and objects are often preferred over tuples.

Objects are the more versatile construct in JSON, possibly with mandatory, optional, and unknown properties. In JSON Model, properties are simply properties, each property name mapped to its type. The first character is used to express whether the property is mandatory (`!`, the default) or optional (`?`). Optional properties can also be described with references and regular expressions, as discussed in the next section. Open documents are allowed by using a special `""` catch-all property, which is consistent with its usage to express any string value. Mandatory and optional constant properties are directly validated. For other properties, first *all* optional references are tried: if the property name matches then the corresponding value must match as well. If properties did not match, *all* optional regex are tried. If not match again, the catch-all case is finally applied.

```
{
  "name": "",
  "age": 0,
  "?friends": [ "" ]
}
```

Figure 6: Tight JSON model for Figure 1

Figure 6 shows a tight model for the data in Figure 1. The 3 expected properties are represented as 3 properties. As `friends` is optional, the property name is prefixed with a `?`. This tight model is slightly shorter than, and quite similar to, the sample value.

3.3 Model Constants and Types

Constants from different types are often needed in data structures, for instance for declaring an enumeration.

Non-empty constant strings represent themselves, so `"Susie"` means string `"Susie"`. However, all non-alphabetical first characters are reserved as a generic mechanism for extensions, e.g. `? !` for property names in Section 3.2 and `= $ ^` which are defined below. We use character `_` (underscore) as an escape prefix, thus `"_Susie"` is also string `"Susie"`, `"_|"` is simple string `"|"`, and `"_"` is the empty string, distinct from `""` which means any string. Other basic constants use character `=` (equal) followed by the expected value: `"=null"` is the `null` value, `"=false"` the boolean `false`, `"=0"` the integer `0`, and `"=6.02E23"` the Avogadro number.

Character `$` (dollar) is a generic entry point to refer to predefined or locally defined types. As a convention, predefined values are in upper case. They include `$ANY` for any possible value, `$NONE` for no possible value, `$U32`, `$U64`, `$I32`, `$I64`, `$F32` and `$F64` for unsigned/signed integers and floats in their 32-bit and 64-bit variants, `$REGEX` for valid regular expressions, `$URI` for URIs, `$DATE` for dates, and others… This mechanism allows to add more formatting or type constraints. Such references can also be used for constraining optional property names, provided that the value is a string.

A common use-case is to constrain string patterns with regular expressions. Character `^` (caret) introduces regular expressions, e.g. `"^[A-Z][a-z]*$"` means any capitalized word in the Latin alphabet. The caret is to be understood both as a sentinel for signaling a regular expression *and* the metacharacter marking the starting position within the string. We recommend that valid regular expressions should be restricted to *safe* regular expressions, excluding inefficient backtracking implementations [13]. These regular expressions can also be used as optional property names, matching any property of that name if it was not already matched.

```
{
  "character": "^(Calvin|Susie)$",
  "pi": "=3.141592741012573242187 5",
  "birth": "$DATE",
  "^(Mon|Tue|Wed|Thu|Fri)$": true,
  "forty-two": "=42",
  "empty-string": "_",
  "$URI": "",
  "": 0
}
```

Figure 7: Constants and Types

Figure 7 presents a model for an object with 5 mandatory properties: Property `character` matches a regular expression, `forty-two` is the integer 42, `pi` is the float π, `empty-string` is the empty string, `birth` is a date. Optional properties are allowed: URI property names must have string values, property names matching the week-day regex must have booleans, other properties must have integers.

3.4 Model Constraints

A great deal of JSON Schema syntax is dedicated to validation constraints on values or sizes. In our opinion, this feature is not an important requirement for data structures, thus it is not prioritized.

Constraints are represented as an object with the specific Property `@` (at sign) to denote the target type, and a limited number of additional properties to describe constraints. The keyword properties `ge gt le lt eq ne` express length or order constraints, their interpretation depends on the values and types: If the target type is an array, an object or string and the value is an integer, the constraint is on the array size, number of properties or string length respectively; If both target type and value are strings, the constraint is a string lexicographic comparison. Optional constraints can be declared with additional keywords, e.g. `distinct` tells whether values are to be distinct in an array, or characters are distinct in a string. Unknown or unapplicable properties must be rejected.

In Figure 8, Model (a) is an array of 42 unique strings, Model (b) is a lower-case word of length between 8 and 10, and Model (c) is a date in May'23.

```
{
 "@": [ "" ],
 "eq": 42,
 "distinct": true
}
```

(a) Array

```
{
 "@": "^[a-z]*$",
 "ge": 8,
 "le": 10
}
```

(b) String

```
{
 "@": "$DATE",
 "ge": "2023-05-01",
 "le": "2023-05-31"
}
```

(c) Date

Figure 8: Constraints with `@`

3.5 Model Composition

With these building blocks, data structures can be constructed from the basic types and by combining objects and arrays. More detailed operators can be useful to factor out common elements, as exemplified by JSON Schema: *and* (`allOf`), *or* (`anyOf`) and *exclusive-or* (`oneOf`). JSON Model uses objects with specific properties (one at a time) applied to a list of models for these operations. The one-at-a-time constraint simplifies the semantics and intelligibility of composed objects.

The most fundamental construct is *or* to express a union or tagged-union, or to list the values of an enumeration. We use Property `|` (pipe) to express this. The generalized *exclusive-or* (*xor*) is a particular case of *or* where only one case must match, which may required implementations to check all alternatives. We use Property `^` (caret) for this. Finally, it may be useful in some (corner) cases to check that a value matches several constraints simultaneously. We use Property `&` (ampersand) for this.

```
{
  "season": { "|": ["Spring", "Summer", "Fall", "Winter"] },
  "movie": {
    "^": [
      { "lang": "français", "titre": "" },
      { "lang": "english", "title": "" },
      { "lang": "Deutsch", "Titel": "" }
    ]
} }
```

Figure 9: JSON Model with Combinations

Figure 9 illustrates *or* and *xor* combinations with an object with two mandatory properties. Property `season` is one of the four strings. Property `movie` is a tagged union, with Property `lang` as a discriminator, to provide the title of a movie in some language.

In practice, the *and* operation is not very useful with tight models because each submodel cannot be extended by the very definition of tightness. The typical use case is to build an object with different sets of already defined properties, much like multiple inheritance in an object-oriented programming language. To address this common use case, we introduce the *plus* operator with Property `+` (plus) which must be understood as a pre-processing macro to combine properties from different objects, which are then interpreted as a standard object. This operator is distributive over `|`. Precise rules define how properties are merged when they have the same name: If one is mandatory and another is optional, then the result is mandatory, and the values must be compatible. Then optional reference and regex properties are merged, then the catch-all property.

Figure 10 Model (a) illustrates the merge of two objects definitions. In the resulting merged Model (b), each object contributes to

```
{
  "+": [
    { "!name": "",
      "?cel": "", "^[a-z]+$": "" },
    { "!cel": "",  "?tel": "", "": 0 }
  ]
}
```

(a) Merging…

```
{
  "!name": "",
  "!cel": "",
  "?tel": "",
  "^[a-z]+$": "",
  "": 0
}
```

(b) Merged

Figure 10: JSON Model Merge Composition

one mandatory property (`name` and `cel`). First object optional Property `cel` becomes mandatory because of the merge rule with `cel` in the second object. The `tel` property remains optional. Properties matching the regular expression must be strings (1st object). Other properties are also allowed if they are integers (2nd object).

```
{
  "name": "Calvin",
  "cel": "060708",
  "desk": "R.02",
  "Age": 6
}
```

(a) Calvin

```
{
  "name": "Susie",
  "cel": "061234",
  "tel": "012345",
  "AGE": 7
}
```

(b) Susie

```
{
  "name": "Hobbes",
  "tel": "010101",
  "CEL": "06",
  "age": 6
}
```

(c) Hobbes

Figure 11: Three Sample Values

In Figure 11, the two first values conform to the merged model, the last does not because mandatory `cel` is missing and `age` is lower case so should be a string instead of an integer.

3.6 Model Complements

This section addresses additional features useful when describing a data structure, such as meta-data, defining and reusing elements and importing external definitions.

A common use case is to add comments in a data structure description to help understanding and maintenance. As JSON does not have a syntax for comments, we use Property `#` (sharp) which will be ignored by model validators. Its value may be a simple string or an arbitrary object, allowing model designers to add any information they see fit.

When developing a large model, it is useful to factor out parts that can be reused consistently, avoiding repetitions. As we already have a mechanism for referencing a model with a string (`$`), we only need to add new definitions inside a model. We use Property `%` (percent) to introduce an object with keys as identifiers and values as the expected corresponding model. The name space is global by default, i.e. multiple definitions of a given name override one another, but can be extended as discussed in Section 4. As a convention, predefined names are in uppercase, and user-defined names are expected to use lower case or be capitalized. To allow simple recursive structures, naming an element can also be done directly in its definition with Property `$` (dollar) and the name as a value.

Figure 12 is a recursive data model. Definition `section` uses itself as an array item in the `sections` property type. The recursion is well-founded because Property `sections` is optional, or the list can be empty. Figure 13 shows a conforming value.

```
{
  "#": "A Book",
  "%": {
    "section": {
      "title": "^.",
      "?text": "",
      "?sections": [ "$section" ]
    }
  },
  "+": [
    { "authors": [ "^." ], "publisher": "^." },
    "$section"
  ]
}
```

Figure 12: JSON Model for a Book

```
{
  "title": "JSON Model: A Description Language for JSON",
  "authors": ["Calvin", "Susie"],
  "publisher": "Hobbes",
  "sections": [
    {"title": "Introduction", "text": "The JSON ..."},
    {"title": "Related Work", "text": "As data ..."}
  ]
}
```

Figure 13: Example Book Data

For large data structures, and to reuse already defined models, it is interesting to be able to reference a model which is stored outside of the model definition. This can be achieved by allowing URLs in **$**-references, including a possible fragment, similarly to JSON Schema, e.g. **$https://json-model.org/geom#polygon** references the **polygon** element type defined with **%** in the referenced model. It is also possible to specify a path to a submodel.

```
{
  "#": "Definitions at https://json-model.org/geom",
  "%": {
    "coord": { "x": 0.0, "y": 0.0 },
    "segment": [ "$coord", "$coord" ],
    "polygon": [ "$coord" ]
} }
```

Figure 14: Geometric Definitions

```
{
  "%": { "geo": "$https://json-model.org/geom" },
  "pol": "$geo#polygon",
  "seg": "$geo#/%/segment"
}
```

Figure 15: Geometric Use

A definition can be used indirectly, as illustrated in Figures 14 and 15. Geometric types are defined in the `geom` model, and then imported and used in another model.

Figure 16 is a self-validating tight meta-model for JSON Model. It does not use any predefs, but redefines **any** and **none** special values. The definition of `Obj` excludes special properties from appearing directly in objects. This meta model only uses disjunctions and

```
{
  "#": "Self-Validating JSON Model",
  "$": "Model",
  "%": {
    "val": { "|":  [ null, true, 0, 0.0, "" ] },
    "any": { "|": [ "$val", [ "$any" ], { "": "$any" } ] },
    "none": { "|": [ ] },
    "meta": { "|": [ "", { "": "$any" } ] },
    "Array": [ "$Model" ],
    "Constraint": {
      "_@": "$Model",
      "^(le|ge|lt|gt)$": { "|": [ 0, 0.0, "" ] },
      "^(eq|ne)$": "$val",
      "?distinct": true
    },
    "Or": { "_|": [ "$Model" ] },
    "And": { "_&": [ "$Model" ] },
    "Xor": { "_^": [ "$Model" ] },
    "Add": { "_+": [ "$Model" ] },
    "Obj": { "": "$Model", "^[|@&^+]$": "$none" },
    "Elem": {
      "+": [
        { "?$": "", "?#": "$meta", "?%": { "": "$Model" } },
        { "|": [ "$Constraint", "$Or", "$And",
                 "$Xor", "$Add", "$Obj" ] }
      ]
    }
  },
  "|": [ "$val", "$Array", "$Elem" ]
}
```

Figure 16: JSON Model Self-Validating Tight Meta-Model

one merge, and is recursive on *Model* and *any*. It could be made tighter, e.g. by using regular expressions to restricts constants, local definition identifiers or list predefs.

```
{
  "#": "A JSON Model Extension",
  "%": {
    "model": "$https://json-model.org/model",
    "ExtendedConstraint": {
      "+": [
       "$model#Constraint",
       { "?mo": "$model#val", "?in": "$model" }
      ]
  } },
  "|": [ "$model", "$ExtendedConstraint" ]
}
```

Figure 17: JSON Model Extension

External references offer a simple mechanism to extend a model, as shown in Figure 17. The new model re-uses JSON Model meta-model to add two new properties to constrained elements.

4 DISCUSSION

This section discusses various aspects of JSON Model, including possible extensions, alternatives and open questions.

JSON Model relies on 15 special keywords on top of JSON syntax, which is much lower than JSON Schema (60) and smaller than

JSound (19). It uses symbols with well-known semantics from programming languages, such as `#` for comments, `| & + ^` for combinations, `$` for references. Ambiguities are avoided because keyword operators (`@ | & + ^`) are exclusive inside elements. Keywords are short to avoid mistyping: they are either one-character symbols (`# @ |`…) or very short mnemonics (`eq ne le`…). These latter keywords could instead rely on symbols unlikely to be used as property names in typical data structures, e.g. `>=` for `ge` or `!` for `distinct`. However, the resulting readability is debatable.

Operator `+` semantics could be enhanced to address more use cases: The value compatibility when merging could cover simple type inclusions. Though it would seem to be interesting, this operator cannot be distributive over `& ^` as it would lead to contradictions.

JSON Model meta data are not standardized in any way. It may help tools to constrain some keywords inside a `#` property. For instance, a model version identification may be useful. Another special usage for meta data would be to deal with validation options, e.g. whether `1.0` should be considered an integer or not.

JSON Model definitions (`%`) are set in a global name space. It should be decided whether collisions override identifiers or are prohibited. Another choice would be to scope definitions inside the element in which they are declared. A simple alternative approach could be to restrict `%` usage to the model root.

Although constraints on integer values are not often used, in most cases they are about positivity (≥ 0 or > 0). This could be embedded directly in the type value, with `0` for positive, `1` for strictly positive and `-1` for any integer.

Sentinel Character `$` semantics is overloaded for predefined types, local and external references. Although this overloading seems harmless, it could help readability to use a different sentinel character depending on the function, e.g. `:` for predefined types, `$` for local references and `$$` for external references.

JSON Path is a standard query path for JSON to access an element within a JSON structure. JSON Schema uses a URL path instead when dealing with references, which is consistent because schemas are identified by URLs. JSON Model does the same to follow suit, but it may make sense to consider a JSON path alternative syntax. Also, it is unclear whether references should be URL-encoded.

Regular expressions are introduced with the `^` character sentinel. Another option would be to surround them between `/`, similarly to Sed, Awk or Perl. There would be two benefits: The regular expression loses the constraint that it starts at the beginning of the string, and it would allow to add options after the second slash.

Compared to JSON Schema, JSON Model left out a few features. Some constraints about types contained in arrays, or about values (multiple-of, mime-type), are removed because field studies show that they are seldom used [3, 15]. They could be added easily with more constraint keywords (e.g. `in mo fmt`), or special definitions (e.g. `$MIME:text/xml`). Type declaration logic keywords `if then else not` are also removed: Not only is this feature seldom used, but it creates very hard to understand schemas and can often be removed. Variadic tuples are not currently covered, but could be with minimal conventions in constrained elements.

It is interesting to investigate what happens if a random JSON object value is loaded in place of a model. This value is likely to be a valid model, because models use type inference to define types, and model-specific symbolic keywords are unlikely to be found in a typical value. However, a value considered as a model is very tight and will probably be only able to validate itself, as most string values will be considered as constants to be checked. Validating any other value would raise an error.

We have developed a proof-of-concept implementation of JSON Model in Python, available online [7]. It offers both a model validator (interpreter) and a compiler which generates more efficient check functions for validating values. The compilation of a validation function from a model provides optimization opportunities, e.g. tagged-unions can be recognized to generate a direct test on the tag property instead of checking each case in sequence. This tool has been used extensively to validate schemas against all versions of JSON Schema [15]. The implementation also includes proof-of-concept model-to-schema and schema-to-model converters.

5 CONCLUSION

JSON Model satisfies the 10 criteria set for its design: It is self-hosted, compact, convenient, frugal, tight, expressive, composable, efficient and extensible as defined in Section 3.1. As for intelligibility, it is judgemental in nature: We think that model illustrations in Section 3 can be broadly understood by a developer even without knowing in advance the formal syntax.

Future work includes completing JSON Model design hopefully with feedbacks from this introduction, writing a formal specification, developing Python and JavaScript reference implementations with extensive positive and negative tests as well as convenient tools such as converters, distributing these productions widely with an illustrated tutorial and example use cases, and investigating the integration of JSON Model into standards such as OpenAPI.

Thanks to O. Hermant for his help in proofreading this paper.

REFERENCES

[1] Cesar Andrei, Daniela Florescu, Ghislain Fourny, Jonathan Robie, and Pavel Velikhov. 2018. JSound 2.0 – The Complete Reference. https://jsound-spec.org

[2] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023. Validation of Modern JSON Schema: Formalization and Complexity. (March 2023). working paper or preprint.

[3] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents. In *40th Int. Conf. on Conceptual Modeling ER 2021 (Lecture Notes in Computer Science, Vol. 13011)*. Springer, 102–112.

[4] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. *Negation-Closure for JSON Schema.* Preprint. https://arxiv.org/abs/2202.13434v1

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conf. on Object-Oriented Programming*. Springer, 257–281.

[6] Juergen Boldt. 1995. *The Common Object Request Broker: Architecture and Specification.* Specification Formal/97-02-25. Object Management Group.

[7] Fabien Coelho and Claire Yannou-Medrala. 2023. JSON Model. https://github.com/clairey-zx81/json-model

[8] Douglas Crockford. 2006. *The application/json Media Type for JavaScript Object Notation (JSON).* RFC 4627. IETF.

[9] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *PoPL*. 207–212.

[10] ECMA International. 2011. *Standard ECMA-262 – ECMAScript Language Specification* (5.1 ed.). 846 pages. First edition in 1999.

[11] Pierre Jouvelot and Rémi Triolet. 1989. *NewGen: A Language-Independent Program Generator.* EMP-CRI 191. CRI, Mines Paris – PSL.

[12] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. 2003. The XML Web: a First Study. In *The Web Conference*.

[13] Mitre. 2021. *Inefficient Regular Expression Complexity.* CWE 1333.

[14] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. *JSON Schema: A Media Type for Describing JSON Documents.* Draft 2020-12. IETF.

[15] Claire Yannou-Medrala and Fabien Coelho. 2023. *An Analysis of Defects in Public JSON Schemas.* Tech. Report A/794/CRI. CRI, Mines Paris – PSL.