JSON Model:
a Lightweight Featureful DSL for JSON

Fabien Coelho and Claire Yannou-Medrala

Centre de recherche en informatique, Mines Paris – PSL, France

Abstract. JSON is a *de facto* standard cross-language textual format used to represent, exchange and store structured data. Data schemas allow to document structures and verify values. Three JSON-based schema description languages have been proposed for JSON so far: JSON Schema, JSound and JSound-C. These languages are verbose and use a lax validation semantics: 60% of public schemas have been found defective [18] because JSON Schema is particularly error-prone. We present JSON Model, an alternative to previous proposals which is both lightweight and featureful, discuss key design choices, its efficient implementation, and its possible integration with other standards such as OpenAPI.

Keywords: JSON; JSON Model; Schema description language; DSL; OpenAPI.

1 Introduction

JSON [10] *JavaScript Object Notation* has become an ubiquitous cross-language *de facto* standard to represent, exchange and store data between computer applications. Its success stems from the extensive use of JavaScript [12] in web applications, and that many services rely on this technology. 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 (list, tuple, sequence, set) and objects (struct, record, dict, map, association, key-value pairs). However, it has a limited number of types, no syntax for comments, unbounded but incomplete numbers (e.g. no *NaN*). Object properties (key, attribute, field) are restricted to strings. Only tree structures can be serialized: 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 including Python, Java, Shell, and SQL.

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

Fig. 1: A JSON object with 3 properties

JSON data are mostly generated and processed automatically from a programming language: Humans 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 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.

This paper presents the design and outlines the implementation and integration 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 introduces our proof-of-concept implementation, including preprocessing and compiler optimizations. Section 5 outlines its possible integration into OpenAPI, before concluding in Section 6.

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 [12]. When considering cross-language features such as data interchange, language-independent description languages can been used, for instance SQL, Newgen [14] or CORBA [6] IDL. 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, and to do data description separately, e.g. with a DTD or XSD for XML, and with JSON Schema, JSound or JSound-C for JSON.

JSON Schema [17] has been under development for 10 versions over nearly 15 years. The latest proposal (2020-12) defines 60 keywords to describe JSON structures, much like classic data structures, but also includes constraints on element sizes (strings, arrays...), uniqueness properties, regular expressions on property names, schema composition (`allOf anyOf oneOf`) and logical assertions (`if then else not`). Its open-document mindset means that schemas are loose by default, allowing any type or property unless otherwise stated. There is a meta-schema which can validate itself. Figure 2 tight schema is suitable for Figure 1 data and is 4 times larger than this sample. The overall complexity [4, 2, 18]

```
{ "$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 }
```

Fig. 2: Tight JSON Schema for Figure 1

```
{ "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 } ] }
```

Fig. 3: Tight JSound for Figure 1

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

Fig. 4: Loose JSound-C for Figure 1

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 proposed to describe JSON and TYSON (a typed extension of JSON) data structures (Figure 3). It is inspired by XML Schema and simpler than JSON Schema. It addresses some of its shortcomings, e.g. types are mandatory, and extensive types appropriate to programming are available. Although some JSON Schema features (composition operators, property name constraints) are not available, it provides inheritance. It shares the same open-document mindset with optional properties by default. JSound comes with a compact variant called JSound-C (Figure 4), which cannot be mixed with full JSound. The whole description syntax must be switched if any feature is not available, such as tight object (mandatory properties) definitions.

We propose an alternate JSON system for describing JSON data which is more compact than JSound-C, but still retains most features from JSON Schema.

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 at REST API by web or mobile applications are typically not-deeply-nested objects with few properties. Data structures are mostly tight, because from a programming perspective it is easier to write code when assumptions can be made about actual types. For this use case, 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 features 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.
intuitive	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	allow to add new concepts if needed.
universal	human and computer language independent.
distinguishable	the description is easy to separate from what is described.

To achieve the *universal* and *distinguishable* features when properties are used in the description language, they are always prefixed with a symbol such as `^ _ $ & | = +`. This choice ensures the language independence of models, and that alphabetical properties described are distinct from modelling properties.

3.2 Model Basics: Type Inference

In order to have a compact and intuitive representation, JSON Model relies on type inference [11], so that each type is represented by a 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 (`true` or `false`), `-1 0 1` are integers (signed, ≥ 0 and > 0), `-1.0 0.0 1.0` are numbers (any, positive, strictly positive), `""` (empty string) is a string, `[]` and `{}` are *empty* arrays and objects. The special definition of positive values stems from the observation that in most cases numbers are really expected to be positive, even if this is rarely declared. A notable exception is geospacial data (e.g. GeoJSON).

As arrays are often used to hold extended homogeneous lists or fixed tuples of heterogeneous items, these two usage patterns are prioritized. A list is expressed as an array of **one** type, e.g. `[ 1 ]` is a list of strictly positive integers. A tuple is expressed as an array of **two or more** types: `[ "", true, [ 0.0 ] ]` is a tuple containing a string, a boolean, and a list of positive numbers. Although this induces a non uniformity as a tuple of one element cannot be expressed, this degenerate case does not seem worth preserving, as the homogeneous list use case is frequent and objects are generally preferred over tuples.

Objects are the most versatile JSON constructs, possibly with mandatory, optional, and unknown properties. JSON Model simply expresses properties as properties, each property name mapped to its type. The first character is used to express whether the constant 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 through the special `""` catch-all property, which is consistent with its interpretation as 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. Finally, the catch-all property is applied.

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

Fig. 5: Tight JSON model for Figure 1

Figure 5 shows a tight model for the Figure 1 object with 3 properties. 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, e.g. for declaring an enumeration. In JSON Model, non-empty constant strings which start with an alphabetical character represent themselves, thus `"Susie"` means `"Susie"`. 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. Character `_` (underscore) is the 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 `=` (equal) followed by the expected value: `"=null"` is the `null` value, `"=false"` is boolean `false`, `"=0"` is integer `0`, and `"=6.02E23"` is the Avogadro number.

Character `$` (dollar) is a generic entry point for references to predefined or locally defined models or types. As a convention, predefined values are in upper case. They include `$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... As a convenience, all basic types have equivalent explicit names: `$STRING` for `""`, `$BOOL` for `true`, `$INTEGER` for `-1`, `$NUMBER` for `-1.0`, `$NULL` for `null`. This is useful for contexts where a very short model would seem awkward, as discussed in Section 5. 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 `/` (slash) introduces a regex which must end with `/` followed by options, e.g. `"/^[a-z]*$/i"` means all case-insensitive words in the Latin alphabet. Only *safe* regular expressions are allowed by default, excluding inefficient backtracking implementations [15]. Implementation may optionally allow unsafe regex. Regex can be used as optional property names, matching any property of that name if it was not already matched.

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

Fig. 6: Constants and Types

Figure 6 presents a model for an object with 5 mandatory properties: Property `character` matches a regex, `forty-two` is integer 42, `pi` is float π, `empty-string` is the empty string, `birth` is a valid date. Optional properties are allowed: URI properties are strings, week-day properties are booleans, and all other properties are integers.

3.4 Model Constraints

JSON Schema has a dedicated syntax to validate constraints on values or sizes. We think that such a feature is not an important requirement for data structures, thus it is not prioritized. Constraints are represented as an object with Property `@` (at sign) to denote the target type, and a limited number of additional properties to describe constraints. Keyword properties `>=` `>` `<=` `<` `=` `!=` express length or order constraints, their interpretation depending on the values and types: For an array, an object or a string and if 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. `!` tells whether values are to be distinct in an array, or characters are distinct in a string. Unknown or unapplicable properties are rejected.

In Figure 7, Model (a) is an array of 42 unique strings, (b) is a lower-case word of length between 8 and 10, and (c) is a date in May'23.

```
{ "@": [ "" ],
  "=": 42,
  "!": true }
```

(a) Array

```
{ "@": "/^[a-z]*$/",
  ">=": 8,
  "<=": 10        }
```

(b) String

```
{ "@": "$DATE",
  ">=": "2023-05-01",
  "<=": "2023-05-31"  }
```

(c) Date

Fig. 7: Constraints with `@`

3.5 Model Composition

With these building blocks, data structures can be constructed from basic types and by combining objects and arrays. The next step is to combine elements with operators to express accumulation (*and*), alternative (*or*, *exclusive-or*) and object merging (*merge*). JSON Model four combinators use objects with specific properties, one at a time, applied to an array of models. The one-at-a-time restriction simplifies the semantics and intelligibility of composed objects.

The *or* combinator uses Property `|` (pipe) to express an union or tagged-union, or to list the values of an enumeration. The generalized *exclusive-or* (*xor*), with Property `^` (caret), is a particular case where only one case must match, which may require implementations to check all alternatives.

```
{ "season": { "|": ["Spring", "Summer", "Fall", "Winter"] },
  "movie": { "^": [ { "lang": "français", "titre":  "" },
                    { "lang": "Íslensk",  "titill": "" },
                    { "lang": "Runasimi", "suti":   "" } ] } }
```

Fig. 8: JSON Model with Combinations

Figure 8 illustrates *or* and *xor* combinators on an object with two properties. Property `season` is one of four strings. Property `movie` is a tagged union with Property `lang` as a discriminator, to provide a movie title. A JSON Model compiler can detect and take advantage of the discriminator to validate values without checking all cases.

It is useful in some rare cases to check that a value matches several constraints simultaneously, which is done with Property `&` (ampersand). However, this operation is useless with tight object models because each submodel cannot be extended by the very definition of tight. The typical use case is rather to build an object with different sets of already defined properties, much like multiple inheritance in an object-oriented programming language. To address this, the merge operator, with Property `+` (plus), allows to combine properties from different objects. It must be understood as a pre-processing macro to merge directly-provided or referenced objects, which are then interpreted as a standard object. This operator is distributive over `^` and `|`, as used by Model `Elem` in Figure 15. Note that it cannot be distributive over `&` as it leads to contradictions. Property definitions are merged when they have the same name **and** the same model value. If one is mandatory and the other optional, the result is mandatory. Then optional reference and regex properties are merged, then the catch-all property. When model values are not equal, merging fails and the model is rejected.

```
{ "+": [
   { "!a": "", "?b": 0, "/^[a-z]+$/": "" },
   { "!b": 0, "?c": "", "": 0 } ] }
```

(a) Merging...

```
{ "!a": "", "!b": 0,
  "?c": "", "": 0,
  "/^[a-z]+$/": ""  }
```

(b) Merged

Fig. 9: JSON Model Merge Composition

Figure 9 Model (a) illustrates the merging of two objects definitions. In the resulting Model (b), each object contributes one mandatory property (`a` and `b`). First object optional Property `b` becomes mandatory because of the merge rule with `b` in the second object. Property `c` remains optional. Properties matching the regular expression must be strings (1st object). Other properties are also allowed if they are integers (2nd object).

```
{ "a": "Calvin",
  "b": 5432,
  "c": "R.03",
  "Age": 6      }
```

(a) Calvin

```
{ "a": "Susie",
  "b": 12345,
  "c": "R.02",
  "AGE": 7     }
```

(b) Susie

```
{ "a": "Hobbes",
  "B": 666,
  "c": "R.07",
  "age": 6      }
```

(c) Hobbes

Fig. 10: Three Sample Values

Figure 10 (a) and (b) conform to the merged model, but (c) does not because mandatory `b` is missing (vs `B`) and `age` is lower case so should be a string.

3.6 Model Complements

This section addresses additional features: meta-data, defining and reusing elements, and importing external definitions.

Comments in a data structure description help understanding and maintenance. As JSON does not have a syntax for comments, we use Property `#` (sharp) for this purpose and for meta-data. Its value may be a string or an arbitrary object, allowing model designers to put any information they need. Property `version`, in the `#` section at the root of the model, must denote the JSON Model version of the model. Validator implementations may use properties in the object values for special configuration purposes, such as whether to consider `1.0` as an integer or not.

```
{ "#": { "name": "A Book", "version": 1 },
  "%": { "Section": { "title": "/./",
                      "?text": "",
                      "?sections": [ "$Section" ] } },
  "+": [ { "authors": [ "/./" ], "publisher": "/./" }, "$Section" ] }
```

Fig. 11: JSON Model for a Book

```
{ "title": "JSON Model: A DSL for JSON",
  "authors": ["Calvin", "Susie"],
  "publisher": "Hobbes",
  "sections": [ {"title": "Introduction", "text": "JSON [10] ..."},
                {"title": "Related Work", "text": "As data st..."} ] }
```

Fig. 12: Example Book Data

When developing large models, 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 (`$`), we only need to add new definitions inside a model. Property `%` (percent), only at the root of the model, introduces an object with properties as identifiers and values as the corresponding model, in a unique namespace. As a convention, as predefined names are in uppercase, these user-defined names are expected to be capitalized, to enhance readability. To allow simple recursive structures, naming an element can also be done directly in its definition with Property `$` (dollar) and the name as a value.

Figure 11 is a recursive data model, where `Section` uses itself as an array item in `sections`. The recursion is well-founded because Property `sections` is optional, or the list can be empty. Figure 12 shows a conforming value.

For large data structures, and to reuse already defined models, it is convenient to reference a model which is stored outside of the model definition. This is 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` model with `%` in the referenced model. Note that, unlike JSON Schema, it is intentionaly not possible to use a path to reference an element in the model: only named elements can be reused.

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

Fig. 13: Geometric Definitions

```
{ "%": { "Geo": "$https://json-model.org/geom" }, "pol": "$Geo#Polygon" }
```

Fig. 14: Geometric Use

```
{ "#": { "name": "Compact Self-Validating JSON Model", "version": 1 },
  "%": {
    "Val": { "^":  [ null, true, -1, -1.0, "" ] },
    "Meta": { "^": [ "", { "": "$ANY" } ] },
    "Array": [ "$Model" ],
    "Cons": { "_@": "$Model",
              "/^(<=|>=|<|>)$/": { "^": [ -1, -1.0, "" ] },
              "/^(=|!=)$/": "$Val", "?!": true },
    "Or": { "_|": "$Array" },
    "And": { "_&": "$Array" },
    "Xor": { "_^": "$Array" },
    "Merge": { "_+": "$Array" },
    "Combi": { "^": [ "$Or", "$And", "$Xor", "$Merge" ] },
    "Obj": { "": "$Model", "/^[|@&^+]$/": "$NONE" },
    "Elem": { "+": [ { "?$": "", "?#": "$Meta" },
                     { "^": [ "$Cons", "$Combi", "$Obj" ] } ] },
    "Model": { "^": [ "$Val", "$Array", "$Elem" ] },
    "Root": { "+": [ { "?%": { "": "$Model" } }, "$Elem" ] }
  },
  "^": [ "$Val", "$Array", "$Root" ]
}
```

Fig. 15: JSON Model Self-Validating Tight Meta-Model

Figures 13 and 14 show a definition used indirectly. Geometric types are defined in the `geom` model, and then imported and used in another model.

Figure 15 is a self-validating tight meta-model for JSON Model. The definition of `Obj` excludes special properties from appearing directly in objects, avoiding ambiguities. This meta model uses disjunctions and two merges, and is recursive on `Model`. It could be made tighter, e.g. by using regular expressions to restricts string constants, local definition identifiers or list predefs.

3.7 JSON Model Summary

JSON Model, introduced in [18, 7], relies on 15 special keywords and 6 sentinel characters on top of JSON syntax, which is much lower than JSON Schema (60). It uses symbols with well-known semantics from programming languages, such as `#` for comments, `| & + ^` for combinators, `$` for references, `/.../` for regex. Keywords are short to avoid mistyping: they use symbols to be language independent and easy to distinguish from the described data structure.

Seldom used features from JSON Schema [3, 18] such as multiple-of, in-array containment and mime-types are discarded. They could be added with more constraint symbols, e.g. `* ~ mime`. Logical assertions are also ignored: This rarely used feature creates very hard to understand schemas and requires expensive evaluations for validation. Further ideas include: Variadic tuples could be covered with minimal conventions in constrained elements; Numbers strictly above 1 could be used to express maximum values.

4 JSON Model Implementation

Proof-of-concept implementations of JSON Model are available [8] in Python and were used extensively in [18]. Compared to JSON Schema, JSON Model simple structure allows to generate definite functions for target elements, without having to care about the external context or keeping track of checked and unchecked elements. This locality feature allows the efficient compilation of models. Our implementation offers three validators: a direct model interpreter; a dynamic compiler which generates check functions on the fly; a static compiler which generates source code for these. Our implementation also includes partial model-to-schema and schema-to-model converters.

The validators share a common preprocessing step: Combinators are flattened to simplify the structure, possibly following `$`-definitions (inlining); Merge operators are distributed over *or* and *xor*-operators and eliminated by computing the resulting objects following property merging rules (Section 3.5); Partial evaluation simplifies empty or one-item combinators and propagates some values when appropriate; When models are demonstrably distinct, *xor* combinators are changed to less costly *or*.

```
{ "%": {
    "A": { "p": 0 },
    "B": { "^": [ "$A", { "q": 0 }] },
    "C": { "+": [ { "r": 0 }, "$B" ] },
    "D": { "^": [ true, { "^": [ 1, 1.0 ] } ] }
  },
  "^": [ "$D",
         { "^": [ "$A", { "|": [] } ] } ]
}
```

(a) Before

```
{ "%": {
    "A": { "p": 0 },
    "B": { "|": [ "$A", { "q": 0 } ] },
    "C": { "|": [ { "r": 0, "p": 0 },
                  { "r": 0, "q": 0 } ] },
    "D": { "|": [ true, 1, 1.0 ] }
  },
  "|": [ true, 1, 1.0, "$A" ]
}
```

(b) After

Fig. 16: JSON Model Preprocessing

Figure 16 illustrates these steps: `C` definition is expanded to remove the merge combinator; `D` combinators are flatten. All *xor* combinators are switched to *or* because the combined models are structurally distinct; the root combinators are simplified thanks to partial evaluation and inlining.

The compilers generate functions for each named types (`%`) and for the target root model, and handles recursion by re-selecting the function when needed. Several optimizations are implemented: Discriminant properties on *or* and *xor* combinators over objects are automatically detected, the generated code inlines the object definitions and takes advantage of the discriminant to call only one object check function; Duplicated models in *xor* combinators are checked once and eliminated directly; `{ "^": [ "$ANY", "$X" ] }` is optimized as model X complement. Enums are detected to generate simple and efficient code. The generated code could be further optimized with partial evaluation. A particular attention is needed to generate accurate messages when a value is rejected, to identify the precise issue and where it occurs in the value *and* the model.

```
{
  "#":
    "float/word",
  "^": [
    0.0,
    "/^[a-z]+$/i"
  ]
}
```

(a) Model

```
import re
re_0 = re.compile("^[a-z]+$", re.IGNORECASE).search
def check_model(val: Any, path: str = "$") -> bool:
    res = isinstance(val, float) and val >= 0.0
    if not res:
        res = isinstance(val, str) and \
            re_0(val) is not None
    return res
```

(b) Generated Code

Fig. 17: JSON Model Static Compilation

Figure 17 illustrates a model and the code generated by our static compiler: The regex is pre-compiled; The code relies on Python dynamic type introspection to handle any value.

5 JSON Model Integration with OpenAPI

Many web and mobile applications use the HTTP protocol to interact with a server, following REST architecture principles [13] to create, read, update, delete and search (CRUDS) data. Swagger then OpenAPI [16] are language-agnostic JSON or YAML descriptions of such interfaces, with JSON Schema [17] used to describe the type of exchanged JSON data. OpenAPI 3.1.0 (2021) extends the latest JSON Schema (2020-12) with some keywords, allows extension properties and defines new formats for usual computer integers and floats. Schema definitions appear as `/components/schemas` in the OpenAPI object, and schemas can be used directly in `MediaType` and `Parameter` objects as `schema` properties, to describe input (request) and output (response) data, either with a reference to a definition or as direct type descriptions.

For assessing qualitatively typical use cases, we have collected 20 API specifications ([19] in `corpus/OpenAPI`) from various sources, ranging from v2.0 to v3.1: 10 API focus on services, with a variety of HTTP methods, and 10 are dedicated to data access (get only), either public open data (6) or private specific data (4). It would be interesting to extend this corpus to study OpenAPI usage. JSON schemas were extracted, analysed [9] and converted automatically to JSON model with our tools [8]. These specs involve non-trivial nested objects and arrays, justifying the need for type specification and documentation, as suggested by frequent meta-data such as `description` and `examples`.

In this sample, format extensions (*int32*...) are frequent. Advance features such as combinators and regex are occasional, and a few schemas include size constraints. Some schemas use many loose objects (unknown or mostly optional properties, null values), but in the context should probably be tight. Confirming our earlier findings [18], the overall schema quality is low: Two files contain fully broken schemas (ebay, walmart), with similar but invalid types and structures, probably generated by the same tool; On the 6 small schemas extracted

```yaml
components:
  schemas:
    Pet:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
      required:
      - id
      - name
      additionalProperties: false
paths:
  /pets/{petId}:
    get:
      ...
```

(a) with JSON Schema (partial)

```yaml
components:
  models:
    Pet:
      id: $I64
      name: $STRING
paths:
  /pets/{petId}:
    get:
      parameters:
      - name: petId
        in: path
        required: true
        model: $I64
      responses:
        '200':
          content:
            application/json:
              model: $Pet
```

(b) with JSON Model (full)

Fig. 18: YAML OpenAPI Comparison (simplified)

from OpenAPI v3 own documentation, 3 have type errors (missing type or invalid format); On the 12 other real-world APIs, only 2 do not have any kind of type errors, and 2 others have minor errors (inconsistent type in examples). Altogether, two-thirds of schemas in our limited sample contain type errors.

JSON Model can be integrated into OpenAPI following JSON Schema approach. As YAML is a superset of JSON, it can be used to represent models. Figure 18 compares OpenAPI with schemas and models. Models are declared in `components/models`, and described in `model` properties. As expected, type definitions are much shorter. For readability, we have used explicit type names instead of relying only on type inference. Moreover, for homogeneity, JSON Schema style references in OpenAPI could use a more direct JSON Model style reference.

6 Conclusion

JSON Model satisfies the criteria set for its design: self-hosted, compact, convenient, frugal, tight, expressive, composable, efficient, extensible, universal and distinguishable as defined in Section 3.1. Intuitiveness is judgemental in nature: We think that model illustrations in Section 3 can be understood by a developer without knowing in advance the formal syntax. The design allows its efficient compilation, and it can be integrated simply in OpenAPI.

Future work includes writing a formal specification, developing a JavaScript reference implementations, and distributing these productions widely with an illustrated tutorial and example use cases. More optimization could be implemented in the preprocessor and the compiler: *or* combinators could reorder models so that more frequent models are checked and matched earlier; model inclusions could be used to simplify some combinators.

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

References

1. Andrei, C., Florescu, D., Fourny, G., Robie, J., Velikhov, P.: JSound 2.0 – The Complete Reference (2018), `https://jsound-spec.org`
2. Attouche, L., Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Validation of Modern JSON Schema: Formalization and Complexity. In: Proceedings of POPL. ACM (Jan 2024)
3. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: 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, pp. 102–112. Springer (Oct 2021)
4. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Negation-closure for JSON Schema. Theoretical Computer Science **955**, 113823 (2023)
5. Bierman, G., Abadi, M., Torgersen, M.: Understanding TypeScript. In: European Conf. on Object-Oriented Programming. pp. 257–281. Springer (2014)
6. Boldt, J.: The Common Object Request Broker: Architecture and Specification. Specification Formal/97-02-25, Object Management Group (Jul 1995)
7. Coelho, F., Yannou-Medrala, C.: JSON Model: a Lightweight Featureful Description Language for JSON Data Structures. Tech. Report A/795/CRI, CRI, Mines Paris – PSL (May 2023)
8. Coelho, F., Yannou-Medrala, C.: JSON Model Implementation (2023–2024), `https://github.com/clairey-zx81/json-model`
9. Coelho, F., Yannou-Medrala, C.: JSON Schema Statistics Tools (2023–2024), `https://github.com/clairey-zx81/json-schema-stats`
10. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF (Jul 2006)
11. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: PoPL. pp. 207–212 (1982)
12. ECMA International: Standard ECMA-262 – ECMAScript Language Specification. 5.1 edn. (June 2011), 846 pages. First edition in 1999
13. Fielding, R.T.: REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine (2000)
14. Jouvelot, P., Triolet, R.: NewGen: A Language-Independent Program Generator. EMP-CRI 191, CRI, Mines Paris – PSL (Jul 1989)
15. Mitre: Inefficient Regular Expression Complexity. CWE 1333 (2021)
16. OAS: OpenAPI. Specification v3.1.0, Linux Foundation (Feb 2021), `https://spec.openapis.org/`
17. Wright, A., Andrews, H., Hutton, B., Dennis, G.: JSON Schema: A Media Type for Describing JSON Documents. Draft 2020-12, IETF (Jun 2022)
18. Yannou-Medrala, C., Coelho, F.: An Analysis of Defects in Public JSON Schemas. In: BDA 2023: 39ème conf. sur la gestion de données – Principes, technologies et applications
19. Yannou-Medrala, C., Coelho, F.: Yet Another JSON Schema Corpus (2023–2024), `https://github.com/clairey-zx81/yac`