Jsonya/dm: A Univocal JSON Interpretation

Miloslav Sredkov
Faculty of Mathematics and Informatics
Sofia University
Sofia, Bulgaria
msredkov@fmi.uni-sofia.bg

Abstract in English—**Despite its popularity as a data interchange format, JSON still lacks a commonly accepted data model and is defined only syntactically. Because of its simplicity, it may appear that the textual representation already conveys all the needed meaning; however, in the context of global Internet-centric applications, where many different technologies interact, the lack of defined semantics can lead to serious interoperability issues.**

In this paper we explicitly attack this problem: we look at the most common JSON interpretations and examine the potential ambiguities of JSON, and then we introduce Jsonya/dm – a strictly defined, language-neutral data model for JSON, which allows consistent interpretation of JSON regardless of the specific environment. To evaluate it, we examine 63 existing JSON libraries for 10 programming languages; the analysis confirms that Jsonya/dm both reflects the established conventions and addresses the potential incompatibilities between these libraries.

Keywords—JSON, information model, data model, interoperability

Abstract in Russian—**Несмотря на свою популярность, формат обмена данных JSON до сих пор не имеет общепринятой модели данных и дефинируется только синтаксически. Из-за его простоты может показаться, что текстовое представление уже передает все необходимые значения. Однако, в контексте глобальных интернет-ориентированных приложений, где множество разных технологий взаимодействуют между собой, отсутствие определенной семантики может привести к серьезным проблемам совместимости.**

В этой статье мы прямо атакуем эту проблему: мы рассматриваем самые популярные интерпретации JSON, исследуем его потенциальные двусмысленности, а затем вводим Jsonya/dm - строго дефинированную, языково-нейтральную модель данных для JSON, предлагающую одинаковую интерпретацию независимо от конкретной среды. Для оценки модели мы рассматриваем 63 существующих JSON библиотек для 10 языков программирования. Анализ подтверждает, что Jsonya/dm одновременно отражает установленные конвенции и исправляет потенциальные несовместимости между этими библиотеками.

Keywords—JSON, информационная модель, модель данных, совместимость

This work was partially supported by the Bulgarian National Science Research Fund through contract 02-102/2009.

I. Introduction

The JavaScript Object Notation (JSON) [1] celebrates a significant growth in popularity and is often applied for the integration of the technologies used in global Internet-centric applications. Because JSON is defined only syntactically, ensuring that all parties interpret it the same way is responsibility of the engineers; for large systems with fuzzy boundaries this can become a real challenge.

What makes it even harder is that most developers implicitly assume “semantics” biased towards the concrete tools they use and fail to observe that others may process JSON differently. Such interpretation clashes may cause interoperability issues when later the system expands. Most approaches to tackle this issue are built around either a specific environment or another data-interchange format; although they suit the particular purpose, none of them defines strict, language-neutral meaning without introducing significant complexity.

In particular, they fail to extend some of the fundamental properties of JSON, namely to be “the intersection of all modern programming languages” and “the thing that everybody can agree on, so it's really easy to pass data back and forth” [2]. This is not a surprise—programming languages and run-time environments are very different, so deriving a data model from one of them is likely to discriminate some of the others. This led us to the following idea: if we want an interpretation which extends the above two principles, then we need to derive it from the specifics of a large enough set of environments. This is what we do in this paper.

To define such a model we started from the syntax of JSON and identified the elements which are prone to multiple interpretations. For each possible meaning we analysed the impact it would have on the different environments, and incorporated the most interoperable ones into the unambiguous data model Jsonya/dm. Most challenging were the interpretation of numbers and the order of object members, but we hope to have achieved acceptable solutions for them. To evaluate our data model we analysed 63 JSON parsing libraries for 10 programming languages and compared the data models they use to ours.

We determined that Jsonya/dm agrees with the design decisions for which the majority of libraries were in unison and provides a reasonable unification otherwise. If applied, Jsonya/dm can bring consistent JSON interpretation to a wide variety of programming languages. As a drawback, if interoperability is less important, some environments may achieve better performance or convenience with a data model suited towards their specific needs.

This paper attempts to resolve the data model ambiguities of JSON while its toolset is still rapidly evolving. Its main contributions are the following:

- an overview of some of the currently used JSON data models (Section II);
- analysis of the ambiguous features of JSON and the general properties of a data model resolving them (Section III);
- the unambiguous JSON data model Jsonya/dm (Section IV);
- assessment of how the data model aligns to current trends based on the analysis of 63 JSON libraries for 10 programming languages (Section V).

II. Existing Approaches

Although to this point no standard JSON data model[1] exists, the lack of such has not prevented numerous successful JSON applications. In this section, we describe some of the data models frequently implied by software engineers.

A. *JavaScript Interpretation*

The origin of JSON have led many people to assume that it should be interpreted the same way as a JavaScript interpreter would do; thus using the data model of JavaScript as a data model of JSON is very common. This raises the question: if JSON is a subset of the ECMAScript Standard [3], should the same be implied for its interpretation?

One of the properties of this data model is that JSON numbers should be interpreted as IEEE 754 [4] 64-bit floating-point values. Examples of this assumption can easily be found around the Internet[2][3][4]. This seems natural for JavaScript-intensive applications, and may also work well for other languages.

Such numeric interpretation, however, can complicate some applications: JSON processing in environments without IEEE-754 floats would be difficult, but a bigger concern is that “since floating point values are converted to and from ASCII representations, we could lose some least significant digits during the translation” [5]. Also, the precision guarantees may be inappropriate for certain applications [6], [7], and even if IEEE-754 floats are appropriate, this data model may still be unsuitable because JSON lacks `+Inf` and `NaN` values.

Treating JSON objects as JavaScript ones also brings an issue. Although the ECMAScript Standard states that “The mechanics and order of enumerating the properties ... is not specified” [3:92], the JavaScript interpreters of many web-browsers enumerate object members in the order of their assignment. Developers unaware that this behaviour is implementation-specific can depend on it and prevent interoperating software from using JSON objects as “unordered collection of zero or more name/value pairs” [1:1]. For environments without an efficient ordered name/value collection like LinkedHashMap[5] this may be problematic.

B. *Metamodel of Another Data-Interchange Format*

JSON is relatively young and its toolset is still developing, so it may be natural to consider it as a simplification of another data format, and to imply the same for its data model. As XML is still ubiquitous, it is often assumed that JSON should be somewhat compatible to it. For example, Wilde and Glushko identify JSON as “an alternative physical model for XML metamodels” [8:48], tools converting between XML and JSON [9], as well as methods to use XML technologies with JSON, including XSLT [10], XQuery [11], [12], and XForms [13], [14] are available.

Employing the information model of XML, however, is not trivial. First, identified by Wilde and Glusko as the “Tree trauma” [15] is the presence of multiple standard metamodels for it. Secondly, due the substantial differences between the two formats, there is no standard way to convert between JSON and XML. Finally, the complexity of XML may prevent JSON from being used as “The Fat-Free Alternative to XML”[6].

YAML, less popular than both XML and JSON[7] but still widely recognised, is another possible alternative. It is stated as a “natural superset of JSON” [16], so many YAML technologies can be applied to JSON too. In addition, its specification explicitly defines the information model of what is available after parsing.

However, YAML is less adopted than XML, so the number of technologies that could be benefited from is limited. It shares some design characteristics with JSON, but is still significantly more complex; with its metamodel the simplicity of JSON would not be taken advantage of. Finally, its metamodel is still loosely defined, e.g.: “The supported range and accuracy depends on the implementation, though 32 bit IEEE floats should be safe.” [16:74], which means that additional negotiations between software engineers may still be necessary.

C. *Other*

1) *Syntax Level Only*

Another approach is to accept JSON only as syntax, so each developer can pick the most suitable data model for their needs. For example, if an application needs to model sequences of command/argument instructions, its developers may choose to interpret objects as ordered multi-maps in order to represent the instructions more conveniently. This is legal because “names

[1] In this paper we use the terms *data model*, *information model*, and *metamodel* as “the constraints on the information entities used to model real world information”.

[2] http://deron.meranda.us/python/comparing_json_modules/

[3] http://blog.mozilla.com/dherman/2011/05/25/a-semantics-for-json/

[4] http://lethargy.org/~jesus/writes/why-json-sucks

[5] http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html

[6] http://www.json.org/fatfree.html

[7] at least in search volume: http://www.google.com/trends/?q=XML,+JSON,+YAML

within an object SHOULD be unique" [1:3], i.e. this is not mandatory.

In fact, working above the syntax level may sometimes not be even needed, e.g. for certain stream processing tasks. The main drawbacks of this approach, however, are that in large systems format specifics like the above one must be explicitly managed in order to ensure that all software components can handle them.

2) *Host Language Types*

Most programming languages have constructs similar to the JSON ones, so many parsers just use the closest possible native data type. This simplifies processing and allows convenient and efficient manipulation.

However, different languages have differences in the corresponding data types, so whether a value is encoded as a string or a number may be irrelevant for PHP and Perl, but is essential for C++ and Java. Other incompatibilities may arise from the meanings of `null` and `false`, associative arrays, etc. Since there are often multiple suitable data types for the same JSON value, incompatibilities may occur even between programs written in the same language.

3) *Custom Object Model*

Finally, a common way to define a metamodel is via an object model defined as a set of custom data types. Many libraries employ this approach; it gives explicit control over the available information and allows easy specification of the information model via the documentation of these types.

In general, this approach does not conflict with ours, and can be used with libraries conforming to Jsonya/dm. Without an explicit language-neutral definition, however, the data model is often influenced by the capabilities of the host language and may be unsuitable for other environments.

III. Analysis

To some readers, JSON may seem sufficiently intuitive and unambiguous. For example, the following JSON code uses all types of values and yet its interpretation seems rather obvious:

```
{
    "name": "Evgeni V. Plushenko",
    "birth_date": {
        "year": 1982,
        "month": 11,
        "day": 3
    },
    "best_scores": [
        261.23,
        91.30,
        176.52
    ],
    "status": {
        "verified": true,
        "locked": false,
        "external_record": null
    }
}
```

Even for this trivial definition, however, some questions can be asked: Would the record be same if we had written `91.3` instead of `91.30`? Could the `external_record` field have been omitted?

For simple web applications the answers to such questions may be irrelevant. This demonstrates one important issue: the ambiguities of JSON are often neglected, because they are not apparent in basic use cases. In this section we will examine the aspects of JSON that may cause inconsistent interpretation with respect to the following styles of representing parsing results:

- *Mutable* data-structures, from which the information is efficiently accessed and manipulated. Predominant in imperative languages, usually based on object properties, associative arrays, or specifically designed data types.
- *Immutable*, and usually purely functional data structures, from which the information can be accessed, and modified versions of it can be efficiently created. Most common in functional languages, but also used in imperative languages for better concurrency or structural sharing.
- *Constant* in-memory representation, from which the information can be efficiently accessed, but no modification mechanism is provided.

In addition, several other aspects must be considered for the data model, including the availability of libraries and system resources, the data model of the environment itself, and whether the information is stored on-disk or in memory.

Although we want to provide a solution appropriate for a wide enough variety of technologies, we cannot consider every possible environment and programming language. For example, paradigms such as logic programming or concatenative programming languages will not be taken into account.

A. *Objects*

The ambiguities of the interpretation of objects, as well as of other elements, are caused by two main uncertainties: which aspects of the JSON representation are essential and which are not, and what values are allowed and what are invalid.

1) *Order of Fields*

One of the most important ambiguities is whether the order of object members, which we will call *fields*, is essential or not. For example, do the two object items in the following example represent identical information?

```
[
    {"a": 1, "b": 2},
    {"b": 2, "a": 1}
]
```

Although RFC 4627 states that objects are unordered, bug reports against libraries not preserving the order can easily be found[8][9][10]. In fact, considering the order of fields essential

[8] https://github.com/flori/json/issues/66

[9] https://github.com/akheron/jansson/issues/15

[10] http://code.google.com/p/json-simple/issues/detail?id=51

allows some problem domains to be conveniently modelled and is useful if JSON files are both human- and machine-processed.

The second reason why unorderness is neglected comes from the possible representations of associative arrays. We identified four most common representations:

- *Plain lists or arrays of pairs* are used in environments where implementation of more sophisticated data structures is unfeasible. Searching a value by name is an $O(N)$ operation, but field order is preserved and can be manipulated.
- *Sorted sequences* provide $O(\log N)$ access without complicated data structures and are suitable for simple *mutable* or *constant* object representation. With structures such as balanced binary trees (or B-Trees for on-disk storage) can also provide $O(\log N)$ modification in both *mutable* and *immutable* setting.
- *Hash tables* are the standard structure in many languages including Java, Python, Perl, and Ruby (up to 1.8.x). They provide $O(1)$ access and manipulation in *mutable* object representation, but do not define any useful traversal order. They are not suitable for *immutable* representations, because the $O(1)$ performance cannot be achieved.
- *Linked hash tables* are hash tables enhanced with additional pointers to maintain the order of elements in a linked-list-like manner. This complication can slightly impact resource usage, but the $O(1)$ performance is still retained. PHP and Ruby 1.9 use them for their associative array constructs, and library implementations are available for most popular programming languages. Like regular hash tables, they are not suitable for immutable setting.

As visible, half of the approaches naturally preserve the order of the fields, and the other half do not. Whether hash tables or linked hash tables should be used is often disputed. Ruby is an example of a language which moved from unordered to ordered hash maps in its version 1.9.1 release[11]. A change in the opposite direction can be seen in Perl 5.8.1, in which 'Mainly due to security reasons, the "random ordering" of hashes has been made even more random'[12][13].

Specifying lack of order guarantees, however, may not be sufficient to prevent developers from inadvertently relying on internal details. Other than the JavaScript example we already mentioned, the documentation of the `Dictionary` generic class from .NET states that "The order in which the items are returned is undefined"[14], yet, developers have noticed that the CLR implementation uses the insertion order as an order for traversal[15][16]. Not only is this assumption implementation dependent, but it also does not hold if some elements were removed before new ones were inserted.

Even if the order of enumeration looks random, leaving it underspecified breaks the (mostly incorrect) assumption that the application of the same algorithm to the same data produces the same result. In contrast, the other three representations (plan lists, sorted sequences and linked hash tables) do not have such an issue—no matter whether they preserve the order or not, the same program using them would work the same way regardless of the data structure implementation details.

It turns out, that there is no best option: the plain list and linked hash map implementations break the unorderness of the JSON objects, plain hash-maps may lead to inconsistent behaviour, and sorted sequences require $O(\log N)$ access time. In immutable setting, however, sorted sequences have a clear advantage over the rest.

2) Field Uniqueness

The fact that in JSON objects the names are recommended but not required to be unique raises some important questions. For example, do the items of the following array define equivalent objects?

```
[
    {"x": 1, "y": 2},
    {"x": 1, "y": 2, "x": 1},
    {"x": 0, "y": 2, "x": 1},
    {"x": 1, "x": 1, "y": 2}
]
```

Depending on the parser and the used data structures, at least 5 different equivalence configurations are possible, ranging from all distinct to all equal. To avoid these complications, and because many JSON tools cannot handle repeated field names, we will simply consider such JSON files invalid.

Even if all fields have unique names, two other questions arise: are there any restrictions on the names of the fields, and how should they be compared?

As the RFC states merely that the name from the name/value pair is a string, we may assume that no further restrictions are imposed. This means that empty strings and names containing spaces or non-Latin characters can be used as field names. This may break certain attempts to map them to host-language identifiers, or at least require that a fallback mechanism to access fields by strings is provided, but we could not identify any reasonable alternative.

To answer how names should be compared, e.g. whether `"J"` is the same as `"\u004a"` we will analyse the ambiguities of strings in Section III-C. Besides that, one particular issue is whether field names are case-sensitive or not. With risk of discriminating some environments, and because case-sensitivity of Unicode characters is far from trivial, we will assume the more popular convention, i.e. that field names are case sensitive.

[11] http://svn.ruby-lang.org/repos/ruby/tags/v1_9_1_0/NEWS

[12] http://cpansearch.perl.org/src/JHI/perl-5.8.1/pod/perldelta.pod

[13] Pointed out by Marcus Ramberg: http://mjtsai.com/blog/2009/02/05/ordered-hashes-in-ruby-19/#comment-472940

[14] http://msdn.microsoft.com/en-us/library/xfhwa508.aspx

[15] http://stackoverflow.com/q/154307/390389

[16] http://forums.asp.net/t/1267419.aspx/1

B. *Numbers*

As already stated in Section II-A, the lack of number specification beyond the syntax layer is an issue affecting certain applications, and a potential source of interoperability problems. Although JSON numbers have an integral part and optionally a fractional part and an exponent, certain details are not clear:

- Are negative and positive zeros (`-0` and `0`) different?
- Is there, like in the C-like languages a difference between integers and floating point numbers, e.g. are `130` and `130.0` different?
- Are the number of trailing zeroes and the value of the exponent essential, e.g. do `130`, `130.0`, `130.00` and `13e1` encode distinct values?
- Is there a particular precision of the numbers, e.g. can we accurately define 0.123456789012345678901?
- Is there a limit on the range of the numbers, or can they be arbitrarily large?

These questions are most often answered depending on the available host language types. For example, the C library Jansson parses numbers into either `double` or `long` / `long long` depending on whether their textual representation contains a dot or an exponential part[17]. In Java, depending on the passed options, Jackson[18], can parse values into BigDecimal objects, which provide arbitrary precision and retain trailing zeroes, thus `130.0` and `130.00` would be different.

The principle to be the intersection of the popular environments is hardly applicable here. First, if we really want to address the majority of languages, we must probably limit ourselves to 32-bit integers—a range insufficient for many applications. Secondly, if a limit is imposed, the numbers beyond it would have to be encoded as JSON strings, which would not be a very elegant solution.

Another approach would be to state that there are no restrictions, but implementations with limited capabilities may fail to process certain values. In fact, this is inevitably true for other types: some environments may fail to parse non-Latin characters, and most will fail to load strings longer than 2^{32} characters, yet it would be unreasonable to solve these problems by restrictions. Not stating limits, however, is not sufficient; applications still need precision guarantees, so that numbers are not unexpectedly truncated by relaying software components.

C. *Strings*

Although relatively intuitive, strings may still be prone to misinterpretations:

- Does escaping matter, e.g. are `"K"` and `"\u004b"` parsed as different values?
- Can we use invalid Unicode, e.g. standalone surrogate characters, code points larger than 0x10FFFF, or illegal UTF-8 byte sequences?
- Does the fact that characters outside the Basic Multilingual Plane are escaped with two surrogate pairs imply that UTF-16 should be used?
- Are there any additional limits due to interoperability considerations, such as strings to not contain nil (`\u0000`), or more than 2^{31}-1 characters?

Luckily, most of these can be easily dealt with. Related to the last question, Bryan addressed an interesting issue in the JSON Group[19] by pointing out that the RFC implies a limit of 996 octets for JSON strings, due to the stated 8-bit compatibility (as defined in RFC 2045 [17]). Fortunately, this restriction was not intended, and even if it were, it would only affect the UTF-8 encoding.

D. *Other Ambiguities*

Another source of ambiguities may be the interpretation of empty or null-like values. All of the following array items represent some kind of empty value:

```
[false, null, 0, "", {}, []]
```

In certain environments, some of these are traditionally indistinguishable; for example C uses 0 for a false value, in Lisp empty lists are represented as a NIL, in PHP empty regular (indexed) and associative arrays are the same value, etc.

Because some programming languages provide type coercion between numbers and strings, it is natural to also ask whether the same value written as JSON number and JSON string would yield the same result, e.g. `123` and `"123"`.

Finally, the formatting of JSON files (e.g. which values are grouped into a single line) is usually lost after a round-trip encoding. This may not be desirable when the file is edited manually, so it may be appropriate to retain some formatting details in the resulting object representation. The same may be even more valuable if non-standard extensions such as comments are used.

E. *Design Considerations*

Our goal of resolving the listed ambiguities is further refined by the following four main design ideas:

- *Explicitness.* To avoid incompatibilities caused by conflicting assumptions, the metamodel should explicitly and unambiguously define which JSON details are essential and which are not.
- *Determinism.* To achieve reliability, the same JSON text should denote the exact same information regardless of the concrete environment, and any loss of information, including numeric precision, must be controllable.
- *Detail concealment.* To avoid potential incompatibilities, the metamodel structure should not

[17] http://www.digip.org/jansson/doc/2.3/conformance.html#real-vs-integer

[18] http://jackson.codehaus.org/

[19] http://tech.groups.yahoo.com/group/json/message/1795

expose any information not strictly defined as essential.

- *Minimalism.* Following the core JSON principles, only information which is useful to a wide enough set of applications should be included.

For some specific uses these restrictions can be too costly. If less strict semantics are more appropriate (e.g. due to performance reasons), such should be achieved outside the bounds of our metamodel.

IV. Jsonya/dm

Following these ideas we designed Jsonya/dm—an unambiguous information model aiming to provide stable common assumptions between parties communicating with JSON. In this section we define it and describe some of its properties.

A. The Metamodel

The fundamental building block of Jsonya is the information entity called *jsonya element*, or simply *element*, which represents the essential information of a JSON fragment, excluding certain details such as spacing and order of object members. Each *element* can be distinguished as one of the 7 *kinds*: *string*, *decimal*, *object*, *array*, *true*, *false*, or *null*. They correspond to the 7 types of values from http://json.org/: ***string***, ***number***, ***object***, ***array***, ***true***, ***false***, ***null***.

String elements represent Unicode strings, i.e. finite sequences of zero or more Unicode code-points (all valid 1112064 code-points from `U+0000` to `U+D7FF` and from `U+E000` to `U+10FFFF`) [18]. *String elements* do not describe how strings were encoded to bytes or how characters were escaped.

Decimal elements represent the exact values of finite decimal numbers (rational numbers with denominator in the form of $2^N 5^M$) and nothing more. They correspond bijectively to the set of finite decimal numbers, so they cannot contain special values such as 1/3 or positive infinity. The JSON texts ‘`0`’, ‘`-0`’, ‘`0.0`’ and ‘`0e1`’ all correspond to the same *decimal element*.

Object elements are associative arrays whose keys are distinct *strings* and whose values are *elements*. The key-value pairs are referred to as *fields*, and we say that each *object element contains* its values. *Objects* are unordered, so any observable enumeration order should depend only on their keys and values.

Array elements represent finite sequences of zero or more *elements*, which have a non-negative integer *size*, and for each integer *i* in [0, *size* - 1] an element denoted as its *i-th item*. Each *array element contains* its *items*.

True, *false*, and *null elements* represent the `true`, `false` and `null` values respectively. Their only observable information is their *kind*.

All elements are finitely nested, i.e. the *contains* relation forms a finite rooted tree of *elements*. There is no other observable information.

B. Domain Enumerability

Although the above explanation clearly describes the information model, a more formal definition may also be valuable. We considered several possible meta-metamodels to define Jsonya/dm, but none of them seemed suitable for such a bottom-level data model. For this reason will simply define the set of all distinct *elements* by assigning a unique non-negative integer to each of them.

$$\mathbf{element}(n) = \begin{cases} \texttt{null} & \text{if } n = 0 \\ \texttt{false} & \text{if } n = 1 \\ \texttt{true} & \text{if } n = 2 \\ \mathbf{decimal}(k) & \text{if } n-2 \equiv 0 \pmod 4 \\ \mathbf{string}(k) & \text{if } n-2 \equiv 1 \pmod 4 \\ \mathbf{array}(k) & \text{if } n-2 \equiv 2 \pmod 4 \\ \mathbf{object}(0,k) & \text{if } n-2 \equiv 3 \pmod 4 \\ \text{where } k = \left\lfloor \frac{n-2}{4} \right\rfloor \end{cases}$$

$$\mathbf{decimal}(n) = \begin{cases} 0 & \text{if } n = 0 \\ M(x(n-1))\ 10^{E(y(n-1))} & \text{otherwise} \end{cases}$$

$$\mathbf{string}(n) = \begin{cases} \texttt{""} & \text{if } n = 0 \\ \mathbf{char}((n-1) \bmod C) \mathbin{++} \\ \quad \mathbf{string}\left(\left\lfloor \frac{n-1}{C} \right\rfloor\right) & \text{otherwise} \end{cases}$$

$$\mathbf{array}(n) = \begin{cases} \texttt{[]} & \text{if } n = 0 \\ \mathbf{element}(x(n-1)) \mathbin{++} \\ \quad \mathbf{array}(y(n-1)) & \text{otherwise} \end{cases}$$

$$\mathbf{object}(s,n) = \begin{cases} \texttt{\{\}} & \text{if } n = 0 \\ \{\mathbf{string}(s + x(n-1)) : \\ \quad \mathbf{element}(x(y(n-1)))\} \mathbin{++} \\ \quad \mathbf{object}(s + x(n-1) + 1, \\ \quad y(y(n-1))) & \text{otherwise} \end{cases}$$

$$\mathbf{char}(n) = \text{string with the code point } \mathbf{cp}(n)$$

$$\mathbf{cp}(n) = \begin{cases} n & \text{if } n \le \texttt{D7FF}_{(16)} \\ n + \texttt{800}_{(16)} & \text{if } n > \texttt{D7FF}_{(16)} \end{cases}$$

$$x(n) = w(n) - y(n)$$

$$y(n) = n - \frac{w(n)^2 + w(n)}{2}$$

$$w(n) = \left\lfloor \frac{\sqrt{8n+1} - 1}{2} \right\rfloor$$

$$M(n) = 10 \left\lfloor \frac{n}{9} \right\rfloor + n \bmod 9 + 1$$

$$E(n) = \begin{cases} -(n+1)/2, & \text{if } n \text{ is odd} \\ n/2, & \text{if } n \text{ is even} \end{cases}$$

$$C = 1112064 \text{ (all valid Unicode code points)}$$

Figure 1. Bijective function computing elements from indices. Here ++ denotes string and array concatenation and object union.

Figure 1 shows the function **element**(*n*), which, from given index computes its corresponding *element*. For example **element**(1000000) returns `["D", [null], false, true]`.

The function is bijective, i.e. it defines a one-to-one correspondence between the non-negative integers and the set of all Jsonya/dm *elements*. Its inverse function, which we will not include here, also has a similar structure.

The mapping is based on the Cantor pairing function [19], slightly modified to work with non-negative integers instead of with positive ones:

$$\pi(x, y) = \frac{x^2+2xy+y^2+x+3y}{2}$$

The x(n) and y(n) from Figure 1 are its inverse functions.

Most elements are encoded in head/tail manner with the pairing function. Non-zero decimal elements are decomposed to the form of $M 10^E$, where M is a positive integer non-divisible by 10 (without trailing zeroes), and E is an integer. The object fields are sorted by the index of their names—each consecutive name is encoded as the difference to the index of the previous field name. This disallows field name repetition and provides unique (canonical) encoding of objects.

Although the presented encoding scheme can be used for other purposes like the generation of testing values, it was designed merely to define the set of distinct Jsonya *elements*. For practical use it may be modified, e.g. to return human-readable strings more often.

C. *Properties*

The metamodel follows the identified design principles. Strings can contain all valid Unicode code points, because the intersection principle could not be applied to them—the set of characters consistently usable in the majority of environments would be too restrictive (e.g. code points 1 to 127).

The most radical design decision in Jsonya/dm is the way numbers are modelled. As the intersection principle could not be applied here too, decimals were chosen because of their importance for various applications [20], and because they can be encoded to and decoded from text without any loss of information. Their name differs from the name of the corresponding JSON values to convey the narrowed meaning. As an unintended consequence, all the 7 *kinds* of *elements* start with distinct letters, which may in some cases be useful.

Object values are usually used to store struct-like records with fixed set of fields, dictionary-like mappings with homogeneous values, or sometimes hybrids of the two. The information in *jsonya objects* is completely sufficient for these purposes. The intersection principle was applied to rule out the field order from the essential information. The range of possible field names however was not limited, and all distinct *jsonya strings* are acceptable and different, because the commonly usable subset would be too restrictive (e.g. only the characters `[a-z_]`).

Arrays are most often used as lists of homogeneous items, as fixed size tuples, or as hybrids of both. Jsonya/dm reflects their widely accepted meaning and steps further to always allow *arrays* to be distinguished from other *kinds* of *elements*. In fact, empty *arrays*, empty *objects*, empty *strings*, *false* and *null* are explicitly defined as distinct values. Also *strings* containing decimal digits are distinct from *decimals*, and *objects* containing index-like keys, e.g. `{"0": "a", "1": "b"}`, are distinct from *arrays* with the same values, e.g. `["a", "b"]`.

We consider the mapping between JSON and Jsonya/dm straightforward, and therefore omit it from this paper. The defined set of values can be used for other representation formats of JSON-based values, e.g. binary representations, or representations decomposed for more efficient searching.

D. *Impact on Implementations*

Because for some environments, this metamodel may be too sophisticated, we do not state that all conforming parties must fully implement it. Instead, the particular limitations can be negotiated explicitly, and appropriate measures to not distort relayed information must be taken if necessary.

The *elements* most likely to be problematic for certain environments are the *decimals*. To avoid overhead, explicit limits (e.g. up to 15 decimal digits) can be negotiated, or parties that do not perform arithmetic operations but merely relay values can use some text-based in-memory representation.

It turns out, that all essential information, i.e. all information observable from *jsonya elements* is the following:

- the kind of each element (*object*, *array*, *decimal*, *string*, *true*, *false* or *null*);
- for *object* elements: the set of the names (keys) of its member fields;
- for *object* elements: from given *string*, the value of the field with that name;
- for *array* elements: their size, i.e. the number of items they contain;
- for *array* elements: from given index, the *element* at that index;
- for *decimal* elements: the number they represent;
- for *string* elements, the Unicode text they represent.

Because of this, implementing a Jsonya/dm conformant object model can be very simple. The following example represents one possible interface for in-memory Java representation:

```
public interface Element {
    String kind();
    Set<String> keys();
    Element field(String name);
    Element item(int index);
    int size();
    String asString();
    BigDecimal asDecimal();
}
```

E. *Limitations*

The last example also shows that Jsonya/dm does not prescribe exactly how an object model can be designed. This is done purposely in order to keep its definition small and simple. For example, the following questions are not answered:

- How is the unorderness of the `keys()` property going to be achieved? Should it be via a sorted or hashed implementation?
- What will happen if a non-existing field or item is requested, or a method non-applicable for the *element kind* is invoked?
- Java's `BigDecimal` distinguishes equal numbers with different scale, e.g. 12.0 from 12.00. How will this additional information be concealed?

Jsonya/dm also does not define how the "inessential" information can be handled in the cases when such is needed. How the order of fields or the particular formatting can be associated with the *elements* without polluting the object model is left to the tool engineers.

Finally, although the ecosystem of JSON is still maturing, many tools have already reached a relatively stable state. The introduction of a metamodel at this stage is threatened by potential incompatibilities with established technologies, especially ones that intensively rely on the information model such as JSON Schema [21], JSONPath[20], or JSON Pointer [22].

V. Evaluation

To assess how the proposed data model aligns with current trends, we analysed a number of JSON libraries, identified the data models they use, summarised their properties and compared them to Jsonya/dm.

A. *Methodology*

From August to September 2011 we performed the following:

- We selected the 10 most discussed[21] programming languages according to LangPop.com[22], and for each of these languages selected all libraries listed in its corresponding section in http://json.org/.
- We identified the data model of each library by analysing its source code and documentation, and writing experimental programs in order to obtain the following information:
  - How was the JSON information provided, e.g. object model or events?
  - Was the string representation (e.g. character escaping) exposed?
  - What was the supported range of characters?
  - Was the textual representation of numbers (i.e. the exact way they were written in) exposed?
  - Were integers and non-integers treated differently and what was the supported range and precision for integral or non-integral numbers?
  - Was the order of object fields exposed, and what data structure was used to represent objects?
  - Were `false`, `null`, empty objects and empty arrays distinguishable?
  - How was the JSON information modelled, e.g. standard types or custom object model, mutable or immutable types?
- We sent preliminary data of the analysis to the JSON group[23], and made a small number of corrections based on the received feedback.
- We summarised the results and used them to assess the properties of Jsonya/dm in order to identify whether the already established tendencies were captured and whether the areas where libraries are too different were addressed.

B. *Results*

The listed libraries were 72 (C++: 8, C: 11, Java: 21, Python: 4, Haskell: 2, JavaScript: 2, Ruby: 3, C#: 12, PHP: 6, Lisp: 3), of which 9 (C++: 2, C: 2, Java: 3, C#: 2) were excluded for various reasons. Here is the summary of the collected information[24] for the remaining 63 libraries:

- *JSON information:*
  - custom object model: 31,
  - object model of standard types: 22 + 3*,
  - mixed standard/custom types: 3,
  - call-backs or tokens: 3.
- *String representation (escapes):*
  - hiden: 56 + 1*,
  - exposed: 6.
- *Character set:*
  - Unicode: 37 + 15*,
  - Unicode without nil: 2,
  - more limited: 7.
- *Integer or fractional discrimination:*
  - based on the presence of `[.Ee]` character: 37 + 4*,

[20] http://goessner.net/articles/JsonPath/

[21] The most discussed languages instead of the most popular were selected because they covered more paradigms and were likely to gain more popularity in future.

[22] http://langpop.com/#normalizeddiscussion

[23] http://tech.groups.yahoo.com/group/json/message/1751

[24] '*' denotes "usually yes, but with some exceptions"

  - no discrimination: 19,
  - based on the value of the number: 3.
- *Textual representation of numbers:*
  - hidden: 43,
  - partly exposed (e.g. only in certain cases or trailing zeroes only): 13,
  - exposed: 7.
- *Non-integer range and precision:*
  - IEEE 754 64-bit float: 33 + 9*,
  - IEEE 754 32-bit float: 3,
  - manual (textual representation): 7,
  - unlimited decimal: 5 + 5*,
  - unlimited rational: 1.
- *Integer range:*
  - unlimited: 15 + 2*,
  - int64: 10 + 6*,
  - same as fractional: 11,
  - int32/int64[25] depending on the platform: 6,
  - int30/int62 depending on the platform: 1,
  - int32: 2 + 1*,
  - int64 ∪ uint64: 2,
  - manual (textual): 7.
- *Field order:*
  - exposed: 29 + 7* (22: linear structures, 7: linked hash table, 7: platform dependent structures),
  - hidden: 20 + 2* (16: hash tables, 6 sorted structures),
  - partly exposed or depending on the environment: 5.
- *Null-like values:*
  - all distinguished: 51 + 6*,
  - issues to handle or distinguish `null`, `false`, empty arrays or empty objects: 6.
- *Mutability:*
  - mutable: 53,
  - immutable: 4 + 3*,
  - call-backs or tokens: 3.

[25] int*N* and uint*N* denote *N*-bit signed and unsigned integers.

C. Interpretation

The analyses of these libraries in the context of the properties of Jsonya/dm outline the following:

- The discrepancy in number handling justified the seemingly radical approach taken by Jsonya/dm. Unfortunately, most libraries could not handle arbitrarily large numbers, and handling limited decimal numbers accurately requires additional effort.
- Jsonya/dm treats strings in agreement with the majority of libraries. Although some environments did not fully support Unicode, no suitable smaller character set could be identified.
- The fact that more than half of the libraries preserved and exposed the field ordering is worrying, as many developers may have considered this information essential. Still if interoperability is desired, the approach of Jsonya/dm is more appropriate.
- The proposed treatment of special values and field names would cause issues only in small number of libraries.
- As a drawback to our data model, the majority of libraries used mutable object models or object models based on standard system types, but Jsonya/dm does not address how such can be efficiently designed.

D. Threats to Validity

The following may have affected the accuracy of the performed evaluation:

- All libraries were considered equal, although they vary significantly in features, quality and popularity. Therefore, certain libraries may have a much wider influence than others, which was not considered in our survey.
- Some of the libraries may have not been analysed correctly, e.g. used in an incorrect way. We believe the percentage of such errors should be small.
- As several months have passed between the analysis, and the completion of this paper, some of the libraries may have changed the data model they use.

VI. Conclusion

We presented Jsonya/dm—an unambiguous data model for JSON. We analysed some widely used alternatives and outlined some of their deficiencies. We identified the common ambiguities of JSON and discussed how they can be resolved. We presented the data model and defined the set of its *elements* via a bijection with the set of non-negative integers. We discussed its properties and limitations, showing that if modifications are not considered, the interfaces of the adhering object models can be simple. We summarised the properties of the 63 JSON libraries analysed during the evaluation, showing that Jsonya/dm is aligned with established tendencies and attacks the common causes of discrepancy.

While we have already built experimental object models, assessing how easily Jsonya/dm can be implemented in various environments remains a task for the future. Specifically, its impact on performance, code size and ease of use needs to be assessed and appropriate data structures and design guidelines need to be suggested. Although mapping JSON code to Jsonya/dm *elements* is straightforward, a formally defined parser can also be a valuable addition.

We believe that the JSON ecosystem would greatly benefit from an explicitly defined information model like the presented one. It is our hope that Jsonya/dm will be accepted by the JSON community, and we look forward to integration with some of the already developed JSON tools.

References

[1] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627 (Informational), 2006

[2] D. Crockford, "The JSON saga," YUI Theater video, 2009,

[3] ECMA, ECMA-262: ECMAScript Language Specification. 5.1 edn., 2011

[4] IEEE Task P754, IEEE 754-2008, Standard for Floating-Point Arithmetic, 2008

[5] R. Neswold and C. King, "Generation of simple, type-safe messages for inter-task communications," in Proceedings of the 12th International Conference on Accelerator and Large Experimental Physics Control Systems, Kobe, Japan , 2009, pp. 137–139

[6] G. Polhill, L. Izquierdo and N. Gotts, "The ghost in the model (and other effects of floating point arithmetic)," Journal of Artificial Societies and Social Simulation 8(1), 2004

[7] D. Monniaux, "The pitfalls of verifying floating-point computations," ACM Trans. Program. Lang. Syst. 30 (2008) 12:1–12:41

[8] E. Wilde and R.J. Glushko, "Document design matters," Commun. ACM 51 (2008) 43–49

[9] D.A. Lee, "JXON: an architecture for schema and annotation driven JSON/XML bidirectional transformations," in Proceedings of Balisage: The Markup Conference 2011, Montréal, Canada, August 2011

[10] M. Joseph, "XSLT and XPath for JSON — Project 6 Research," https://www.p6r.com/articles/2008/05/06/xslt-and-xpath-for-json/, 2008

[11] R. Bamford, V. Borkar, M. Brantner, P.M. Fischer, D. Florescu, D. Graf, D. Kossmann, T. Kraska, D. Muresan, S. Nasoi and M. Zacharioudakis, "XQuery reloaded," in Proc. VLDB Endow. 2(2), 2009, pp. 1342–1353

[12] J. Robie, M. Brantner, D. Florescu, G. Fourny and T. Westmann, "JSONiq, XQuery for JSON, JSON for XQuery," in XML Prague 2012 – Conference Proceedings, Prague, Czech Republic, 2012, pp. 63–72

[13] A. Couthures, "JSON for XForms, adding JSON support in XForms data instances," in XML Prague 2011 – Conference Proceedings, 2011, pp. 13–24

[14] S. Pemberto: "Treating JSON as a subset of XML," in XML Prague 2012 – Conference Proceedings, Prague, Czech Republic, 2012, pp 81–90

[15] E. Wilde and R.J. Glushko, "XML fever.", Queue 6(6), 2008, pp. 46–53

[16] O. Ben-Kiki, C. Evans and I. döt Net, YAML ain't markup language (YAML™) version 1.2, 3rd edition, patched at 2009-10-01. http://yaml.org/spec/1.2/spec.pdf, 2009

[17] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," RFC 2045 (Draft Standard), 1996

[18] The Unicode Consortium, "Chapter 3 Conformance" in The Unicode Standard, Version 6.1.0, Mountain View, CA, 2012, http://www.unicode.org/versions/Unicode6.1.0/

[19] G. Cantor, "Ein Beitrag zur Mannigfaltigkeitslehre," Journal für die reine und angewandte Mathematik 84, 1878, pp. 242–258

[20] M.F. Cowlishaw, "Decimal floating-point: Algorism for computers," in Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03). ARITH'03, Washington, DC, USA, IEEE Computer Society, 2003, pp 104–111

[21] K. Zyp and G. Court, "A JSON media type for describing the structure and meaning of JSON documents," Internet-Draft draft-zyp-json-schema-03, IETF Secretariat, 2010

[22] P.C. Bryan and K. Zyp, "JSON Pointer," Internet-Draft draft-ietf-appsawg-json-pointer-01, IETF Secretariat, 2012