JENRS

Received: 24 February, 2022, Revised: 20 April 2022, Accepted: 01 May, 2022, Online: 28 May, 2022

DOI: https://dx.doi.org/10.55708/js0105025

# An Algebraic Specification/Schema for JSON

**Konstantinos Barlas**[*,1], **Petros Stefaneas**[2]

[1]Department of Informatics and Computer Engineering, University of West Attica, Athens, 12243, Greece

[2]School of Applied Mathematical and Physical Sciences, National Technical University of Athens, Athens, 15780, Greece

[*]Corresponding author: Konstantinos Barlas, University of West Attica, Ag. Spyridonos Str., Egaleo & kosbarlas@uniwa.gr

**ABSTRACT:** **J**ava**S**cript **O**bject **N**otation (JSON) is an open standard data format that is used widely across the internet as means of exchanging structured data due to its low overhead. While originally created in the early 2000s, it has only gained standard status in 2013 and then again in 2017 with a new version that focused more on security and interoperability. In this paper the authors present a different specification of the JSON standard that relies on algebraic formal methods and provides certain benefits over a regular natural language specification. This specification can also function as a schema that can attest a JSON data document's compliance to its blueprint.

The absorption of Formal Specification methods by the industry happens at a very slow pace, mostly because there is little incentive to tread into a fairly unknown territory. Notwithstanding this reluctance, the authors encourage the usage of Formal Specification techniques to the specifications of open standards; Formal specifications are more succinct, less ambivalent, consistent to the standard, reusable as they support module inheritance and can be executable.

The process of designing new Standards can benefit from Formal Specifications as the resulting specification i) is more tangible; ii) allows a thorough and clear understanding of the standard and also iii) allows property checking and property verification.

**KEYWORDS** Formal Specifications, Algebraic Specifications, Formal Methods, JSON, open standards

## 1. Introduction

Standards released by a standard setting organization are usually accompanied by a publicly-available delineating *specification document* that consists of the requirements that any implementation of the standard has to hold [1]. However, because such a document is written in a natural language there can be some shortcomings:

1. **Verbosity**: Just like any technical document, a specification document written in any natural language often yields quite *lengthy* documents. For instance, the specification of Digital Imaging and Communications in Medicine (DICOM), an open standard that describes the operations that can take place in medical imaging (e.g. handling, storing, printing, and transmitting information) consists of 18 different documents ([2]) and a total of 4900 pages.

2. **Vagueness**: Using natural languages to express terms that depend on being expressed with precision and unambiguity can be an arduous task that usually results in documents that are difficult to comprehend [3]. Words with different meanings or phrases that hypothesize some common background can make this issue even worse. Clarity is important when trying to formalize a statement.

3. **Requirements intermixture**: Due to the non modal nature of natural languages, we often see the merging of a standard's disparate requisites [3]. This renders the tracking of the consequences that a change incurs rather cumbersome, since it is better to examine each requirement individually rather than process a fusion of different yet connected requirements.

4. **Requirements confusion**: Different kinds of a standard's properties can appear mixed up; functional and non functional requirements, system goals and design information are at times used as if they are the same thing [3] .

Those shortcomings can create issues throughout the standard's development and specification stages, issues that are frequently ascertained at the latest stages and are therefore, costly to correct [3]. These drawbacks can be present to any standard's specification document (open or not) but this paper focuses on open standards since their specifications are publicly available.

In this paper we:

1. Reason about how a Formal Specification of an open standard can deal with the issues of specifications written in natural languages that are discussed in this section.

2. Create a Formal Specification of the JSON open standard by converting JSON entities into their equivalent

formal versions.

3. Address the points raised earlier and argue about how this specification methodology is better than the natural language one. In order to do that, a test case will be presented in which we use the formal specification of JSON as a schema, in order to verify additional properties.

This paper is an extended version of [4], a short conference paper that introduced an algebraic schema for JSON and presented the basics of treating JSON objects in an algebraic way. This paper provides the details for almost all operators needed to handle a JSON object as an algebraic item as well as how they operate together and, in an attempt to respond in detail to comments from the conference review, it also expands more on how a formal specification of an open standard can be beneficial to the standard as it can be better than a specification written in natural language. Both this paper as well as its predecessor continue in a similar path as [5], in which the authors have formalized XML structures and used the result to create a formal version of RSS v2.0 open web standard.

## 2. Background material

### *2.1. Open Standards*

While the definition of a Standard can be quite straightforward (an established norm/requirement for a repeatable technical task), it's the term "open" that makes the definition tricky as that word can have a different meaning depending on the point of view; dictionaries, national IT agencies, World Trade Organization (WTO), etc. all provide different points of view for the term "open". An intersection of all those different definitions deals with their motivation, development and usage. The motivation to create and promote an Open Standard usually has to do with promoting interoperability. When it comes to its development, an Open Standard:

1. Most of the times emerges by a process that can be effortless for anyone to attend.
2. Is open to public input [6, 7].
3. Is not regulated by any particular collective or vendor.

#### *2.1.1. Advantages of Open Standards*

Developing and adhering to an open standard does not produce any financial gain, at least not in the short term but in the long run, the interoperability, the sustainable development and the smooth communication channels that come along, all give value to this investment. Additional benefits from following Open Standards are summarized below [8, 9]:

- Open Standards ensure that a vendor cannot gain full control over a format. Since an open standard's specification is available to the public, another party can always implement a solution that adheres to its requirements if the previous solution stopped working for whatever reason.
- Open Standards provide smooth interoperability between different systems that may even use different technologies. Communication between such systems can be difficult if there is not an "agreement" to use a pre-approved channel that each system can identify and work with. For example, a company that requires that all its desks use office software applications compatible with the Open Document format (an Open Standard) gives each employee the freedom of choosing to use **any** such office software, ensuring full compatibility with the rest of the company.
- Open Standards can act as a buffer against applications that stop being developed. If such an application was using a proprietary data format then an end user would have a difficult time trying to save the data in a way that some other application could use. However, if that application was using an open standard document format, it would not be difficult for another application to make use of that data. The issue of program data becoming unusable can be even worse for any big organizations like government agencies (e.g. police [10]) or national Electronic Health Records ([11]).

#### *2.1.2. Problems of Open Standards*

On the other hand, Open Standards have some noteworthy drawbacks. Firstly, creating an Open Standard (with all its steps; designing, reviewing, rating, implementing) can be quite costly in resources. For instance, the amount of time required for an Open Standard to go through that process varies, from a few months to many years, depending on the standard-setting organization.

Also, interoperability can be achieved even with non open standards. There is an abundance of proprietary standards that come with a Reasonable and non-discriminatory licensing (RAND) that do not hinder in any way interoperability (e.g. GSM, CD, DVD, MPEG, Wi-Fi, etc.) [8], [9].

### *2.2. About Algebraic Specifications of Open Standards*

Formal (especially algebraic) specifications can complement (or even substitute) the original natural language specification of the standard as they can deal with the problems discussed in Section 1:

1. Formal specifications are most often than not small in size, mainly due to their modular nature and their basis on mathematics.
2. Their mathematical foundation also allows them to be as precise as mathematics can allow, resulting in unambiguous results.
3. Many specifications can be executable, enabling developers to test an implementation of a standard against its specification.

While this process can be applied to any kind of standard, Open Standards are better candidates for a formal specification since their original specification, the one written in a natural language, is available for anyone to read and attempt to improve by formalizing (versus trying to reverse engineer a proprietary standard). Considering that this process is not done by the standard-setting organization, any attempts to formalize a standard are only as good as the understanding of the standard by the reader. Depending on how intricate and entangled the original specification is, such a task can be of considerable difficulty.

This process may also uncover flaws in the standard; since the formalization process forces someone to ask the right questions, there have been cases in which ambiguities in original natural language specifications have been detected, for example the specification of "time-to-live" element of **R**eally **S**imple **S**yndication (RSS) in [5]. Naturally, the time and resources needed for a formal specification are increased (or differently allocated according to Sommerville [3]), unless the formal specification is developed along the standard.

Formal Methods may at first alienate many readers, especially those not sufficiently versed, however getting acquainted is not such a difficult task [12, 13] and the benefits may make it worthwhile. Perhaps until Formal Methods become more popular, a middle ground would be to have both natural language and formal specifications for standards.

### 2.3. CafeOBJ

The formal specification of JSON that is presented in Section 3 is written using CafeOBJ, which according to its webpage ([14]) is an "advanced formal specification language which inherits many advanced features (e.g. flexible mix-fix syntax, powerful and clear typing system with ordered sorts, parametric modules and views for instantiating the parameters, and module expressions, etc) from 'OBJ' (or more exactly 'OBJ3') algebraic specification language".

The OBJ (and OBJ3) family has been used in many applications ([15]) such as the debugging of algebraic specifications, rapid prototyping, the executable definition of programming languages, the specification of software systems (such as GKS graphics kernel system, Ada configuration manager, Macintosh QuickDraw, etc), specification of hardware, specification of user interface designs and theorem proving. The main axiomatic systems that it uses are order-sorted algebras [16, 17] (used to specify abstract data types) and hidden algebras [16, 18] (used to specify abstract state machines, enabling object-driven specifications).

Algorithm 1 shows a CafeOBJ module for factorials. A specification in CafeOBJ consists of modules; places where sorts, operators and equations are declared [19]. The keyword *mod!* initiates a module and is followed by curly brackets that define the start and end of the module's code block. The next line, *pr(INT)* imports the built-in integer module, allowing sort, operator and equation inheritance. To declare an operator, keyword *op* is used. Operators are like functions and they need explicit domains and codomains. In Algorithm 1, operator *factorial* assigns an integer (domain) to another integer (codomain). A variable in CafeOBJ is declared with the keyword *var* followed by its name and domain. Here, variable *N* belongs to *INT*, the sort of Integers that was imported into the module with the *pr* command. To write the equations describing the factorial we will need a recursive case as well as a base case, as shown in the last two lines of the module, starting with *ceq*. Once the module declaration is over, command *select* selects the created module so that we can use its definitions to calculate the factorial of a number. The command *red* command will try to reduce $fact5$ to its value using the module's equations and left-to-right rewriting rules. Lastly, inline comments in CafeOBJ use the −− prefix.

**Algorithm 1:** Factorial's definition in a CafeOBJ module

```
mod! FACTORIAL
{
    -- Factorial declaration
    pr (INT)
    op factorial : Int -> Int
    var N : Int
    ceq factorial(N) = N * factorial(N - 1) if N > 0 .
    ceq factorial(N) = 1 if not (N > 0) .
}

select FACTORIAL .
red (factorial(5)) .
```

### 2.4. About JSON

**J**ava**S**cript **O**bject **N**otation (**JSON**) is an open standard data format and file interchange format that does not bind itself to any specific language and declares entities as lists of properties in the *attribute: value* format [20]. JSON started as a non-strict subset of JavaScript [20], as defined in the European Computer Manufacturers Association Script (ECMAScript) Programming Language Standard, Third Edition [21], but ever since, it can be (and has been) used in many programming languages. JSON was originally designed to be minimal, portable, textual, a subset of JavaScript [21] but not too far away from the broad spectrum of C-family of languages ([22]).

JSON became a standard originally in 2013, under the name ECMA-404 [23]. The second version of the ECMA-404 standard came along in 2017 [24]. Request for Comments (RFC) published JSON as a standard in 2017 [21] and this is the current JSON standard version that is compliant with ECMA-404 [24].

JSON can express structured data in six ways; the four non-structured primitive types (booleans, strings, numbers and the unique "null" type) and two structured types (arrays and objects) [21]. Arrays and objects can be nested into larger, complex JSON structures, [22].

### 2.5. Uses

JSON has become a very popular data format, with a broad range of applications. It is being used as a configuration language, storing application settings in its low-overhead format, although surprisingly, it lacks support for comments, discouraging people from meddling directly with

such files. JSON is intended as a lightweight alternative to the widespread XML format that has bigger space requirements as XML uses more overhead to store its data.

JSON can be used with most of the languages of the C-family (such as C, C++, C#, Java, JavaScript, Perl) but is also supported by many more, such as Python, Filemaker, .Net, Matlab, Lisp, Haskell, Fortran, OCaml, Prolog, Rust, Scala, etc. [22].

### 2.6. Similar works

Formal versions of JSON schemas have been created in the past, e.g. [25] in which JSON is used as a JSON schema. There also is [26], in which the authors display an Extended Backus–Baur Form (EBNF) grammar of JSON. While JSON and XML are different, formalizing XML is quite similar in nature with the ideas presented in this paper; the authors in [27] have created a formal framework for XML schema languages. What distinguishes this paper from the rest is that a valid schema for JSON is created using an algebraic specification language.

## 3. JSON entities as algebraic objects

The proposed schema for JSON data (Section 3.3) treats JSON objects as lists (Section 3.1) and is written in CafeOBJ (Section 2.3). In order to create an algebraic specification of a JSON file in CafeOBJ, the JSON file has to be converted into a format that CafeOBJ can read and process. A simple command line converter has been created that parses a JSON file and outputs it into a format that we will call JSON-OBJ. The rules for such a translation can be seen in Section 3.2. The resulting JSON-OBJ file can now be loaded into CafeOBJ and can be reasoned with, as seen in Section 3.4.

While the reason for choosing CafeOBJ as the language for this formal schema is the authors' familiarity with it, we would argue that a formal specification of JSON would work equally well with any Formal Method tool; especially those that put more weight on data structures like Z, VDM, Estelle or any language of the OBJ family (e.g. Eqlog, FOOPS, Kumo, Maude, OBJ2, OBJ3). Selecting a proper Formal Method tool can be quite a daunting task as lots of different tools exist, all specializing in different features, yet often having some overlap with other tools.

### 3.1. Lists

JSON describes, in its essence, a list of structured data. A list is a collection of elements where an element can be a list itself. A list module provides operators that can i) append an element into a list, ii) search if an element is contained in one, iii) remove an element from a list and iv) concatenate two lists. These operations are specified in a recursive manner.

For example, Algorithm 2 shows the equations necessary for removing an element from a list. The operator "\" takes an element and list and removes that element from the list, if the list contains that that. The first two equations provide the base cases and the next two the recursive cases. The recursive cases state that given an element $Elem_1$ that's merged with a list $Lst$; if we want to remove an element $Elem_2$ from that list, then if $Elem_1$ is the same as $Elem_2$ then the operation returns $Lst$, otherwise the removal operation is now applied to $Lst$.

**Algorithm 2:** Removing an element from a list

```
op _\_ : List Elt → List
vars Elem1, Elem2 : Elt
var Lst : List
eq (nil  Elem1) = nil .
ceq (Elem1 \ Elem2) = nil if (Elem1 = Elem2) .
ceq ((Elem1 | Lst) \ Elem2) = Lst if (Elem1 = Elem2) .
ceq ((Elem1 | Lst) \ Elem2) = Elem1 | (Lst \ Elem2)
  if not(Elem1 = Elem2) .
```

Algorithm 3 shows the declaration and equations for querying a list for the existence of an element. The operator "*//in*" returns true if a given element is inside a given list. The base cases describe how we check an element against another element or against an empty list. The recursive case states that an element $Elem_1$ exists in a list that consists of an element $Elem_2$ and a sublist $Lst$, only if $Elem_2$ is $Elem_1$ or $Elem_1$ exists in $Lst$.

**Algorithm 3:** Querying a list for the existence of an element

```
op _//in_ : Elt List → Bool
ceq (Elem1 //in Elem2) = true if (Elem1 = Elem2) .
eq (Elem1 //in nil) = false .
ceq (Elem1 //in (Elem2 | Lst)) = true if (Elem1 =
  Elem2) or (Elem1 //in Lst) .
```

### 3.2. JSON data in CafeOBJ format

The most basic JSON element comes in the form of {"key": "value"} and that would get converted into <"key" [ txt( "value" ) ] >. Symbols "<" and ">" contain the element's name inside quotes and its following value inside brackets. Values can be in a number of different types (string, number, boolean, object, array, *null*), so the value of an element is held within the right operator for its type; string values are contained inside the *txt* operator, *log* for boolean values, *int* for integers, etc. A more complex JSON element is the *object*, which is a comma separated collection of *key:value* elements held inside curly brackets. Its equivalent form in CafeOBJ notation is a series of parenthesized elements, concatenated with the @ operator.

The last JSON element type is the *array*, an ordered, comma separated collection of *key:value* elements that is contained within brackets. Since the sequence of the array elements needs to be preserved, the < "key" [ txt( "value" ) ] > format is extended so that the $n^{th}$ element of such an array would be declared as ({ n } <"key" [ txt( "value" ) ] >).

A value in JSON can be another JSON element, as this format supports nested elements. Some sample JSON data can be seen in Algorithm 4 and its conversion to the JSON-OBJ format can be seen in Algorithm 5. Algorithm 5 also shows how each JSON data type gets converted into its appropriate JSON-OBJ format (string - *Name*, boolean - *isAlive*, number - *age*, object - *address*, array - *phoneNumbers* and null - *spouse*).

**Algorithm 4:** Sample JSON data

```
{
    "Name": "John Smith",
    "isAlive": true,
    "age": 27,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumbers": [
        { "number": "212 555-1234",
        "type": "home" },
        { "number": "646 555-4567",
        "type": "office"}
    ],
    "spouse": null
}
```

**Algorithm 5:** Sample JSON data in CafeOBJ format

```
(< "Name" [ txt( "John Smith" ) ] >) @
(< "isAlive" [ log( true ) ] >) @
(< "age" [ int( 27 ) ] >) @
(< "address" [
    (< "streetAddress" [ txt( "21 2nd Street" ) ] >) @
    (< "city" [ txt( "New York" ) ] >) @
    (< "postalCode" [ txt( "10021-3100" ) ] >) @
    (< "state" [ txt( "NY" ) ] >) ] >) @
(< "phoneNumbers" [
    ( 1
        (< "number" [ txt( "212 555-1234" ) ] >) @
        (< "type" [ txt( "home" ) ] >) ) @
    ( 2
        (< "number" [ txt( "646 555-4567" ) ] >) @
        (< "type" [ txt( "office" ) ] >)
    )
] >) @
(< "spouse" [ NULL ] >)
```

### 3.3. Algebraic Schema for JSON

The operators in the algebraic schema for JSON can be employed to parse and validate a JSON-OBJ (as described in Section 3.2) file against its requirements. The operators used in the schema can be seen in Table 1 and are used to extract the data stored (and the way it is organized) in any JSON file. The output can then be used in order to check the file against its specification.

In this paper we will describe operators *jNodeExists* and *getParent*.

Table 1: Operator names and their descriptions.

| | |
|---|---|
| *getKey* | Given an element, it returns its key name |
| *getValue* | Given an element, it returns its value |
| *returnJnode* | Given a list of elements and an element's key, it returns the whole element. If the element is within an array, returns the parent as well |
| *jNodeExists* | Given an element's key name and a list of elements, it searches for the key in the element list and returns a boolean value |
| *hasNextElement* | Given a list of elements and an element's key, it returns a boolean value based on whether the key has a *next* element |
| *getParent* | Given an element's key name and a list of elements, searches for the key name inside the list and returns its parent, if there is one |
| *getArrayNo* | Given an array and an element's key, it returns its array index value |

#### 3.3.1. jNodeExists

Algorithm 6 displays the declaration for the *jNodeExists* operator along with its equations; The operator takes a key name (*KeyName*) and a list of elements (*ElementList*) and returns a true/false value. In the equations that define the operator, $Key_1$ and $Key2$ are variables that describe a key name. *Cont* is a variable that describes content of any sort. $Elem_1$ and $Elem_2$ are variables that describe elements and $N$ is a natural number variable that describes the $N^{th}$ element of an array.

The base-case of the operator's recursive definition is given in the first two equations: if the list of elements is just a single element then either its key name matches the one given to the operator and the operator returns *true*, or there is no match and the operator returns *false*. Two base-case equations are needed considering that *keyname* can also try to match a single element of an array.

**Algorithm 6:** jNodeExists operator

```
op jNodeExists : KeyName ElementList -> Bool
eq jNodeExists(Key1, < Key2 [ Cont ] >) = if (Key1
  == Key2) then true else false fi.
eq jNodeExists(Key1, N < Key2 [ Cont ] >) = if (Key1
  == Key2) then true else false fi .
eq jNodeExists(Key1, Elem1 @ Elem2) = if
  (jNodeExists(Key1, Elem1)) then true else
  jNodeExists(Key1, Elem2) fi .
eq jNodeExists(Key1, < Key2 [ Cont ] > @ Elem1 ) =
  if (Key1 == Key2) then true else jNodeExists(Key1,
  Elem) fi .
eq jNodeExists(Key1, N < Key2 [ Cont ] > @ Elem1 )
  if (Key1 == Key2) then true else jNodeExists(Key1,
  Elem) fi .
```

The recursive actions for the operator are given in the next three equations:

- Given a key name $Key_1$ and an element list that is made out of two elements, the operator will call itself on the first element and if there is no match, it will do the same to the second element.

- Given a key name $Key_1$ and an element with key name $Key_2$ that is concatenated with a list of elements, it will compare $Key_1$ against $Key_2$ and if they are different it will continue recursively with the list of elements until the expression collapses to one of base case equations. Again, two recursive equations are required due to the two different ways an element can present itself; as part of an array or in any other way.

#### 3.3.2. *getParent*

Algorithm 7 displays the declaration for the *getParent* operator along with its equations; The operator takes a key name *K* and a list of elements (*ElementList*) and then seeks element *K* inside *ElementList*, returning the key name of *K*'s "parent", or a *null* value if that element is not found or has not parent. Since this operator relies on *jNodeExists* (Section 3.3.1), this shows the way multiple operators can work together.

In the equations that define the operator, $Key_1$, $Key_2$ and $Key_3$ are variables that describe key names. *Cont* is a variable that describes content of any sort. $Elem_1$ and $Elem_2$ are variables that describe elements.

The first two equations describe the base case: If asked to retrieve the parent of an element named $Key_1$ from a single element *Elem*1, the operator returns null (the special *nil* operator). If asked to retrieve the parent of an element named $Key_1$ from an element called $Key_2$ that contains a single nested element (*Elem*), then it will invoke *jNodeExists*, to search for element $Key_1$ inside *Elem*, returning $K_2$ if found, or *nil* otherwise.

The next three equations describe what may happen if we are searching for the parent of element with key name $Key_1$ in the following scenarios:

- The element list is a concatenation of a simple element and any other element: *getParent* operator ignores the simple element as it cannot have a parent and gets applied to the other element.

- The element list is a concatenation of a simple element and one with nested elements: *getParent* operator ignores again the simple element and gets applied to the element with the nested elements.

- The element list is a concatenation of two elements that both have sub-elements: if the first element contains $Key_1$ then *getParent* returns its key Name otherwise it is applied to the second element.

**Algorithm 7:** getParent operator

```
op getParent : KeyName ElementList -> KeyName
eq getParent(Key1, < Key2 [ Cont ] >) = nilKey .
eq getParent(Key1, < Key2 [ Elem1 ] >) = if
 (jNodeExists(Key1, Elem1)) then Key2 else nilKey fi .
eq getParent(Key1, < Key2 [ Cont ] > @ Elem1) =
 getParent(Key1, Elem1).
eq getParent(Key1, < Key2 [ Cont ] > @ < Key2 [
 Elem2 ] >) = getParent(Key1, < Key2 [ Elem2 ] >) .
eq getParent(Key1, < Key2 [ Elem1 ] > @ < Key2 [
 Elem2 ] >) = if (jNodeExists(Key1, Elem1)) then Key2
 else getParent(Key1, < Key2 [ Elem2 ] >) fi .
```

### 3.4. Test case

This section examines a hypothetical scenario in which a company's management software keeps its contacts in JSON format and those contacts need to adhere to the following traits:

1. Entries for *Name*, *address* and *isAlive* are necessary.
2. *Address* must have entries for *streetAddress*, *city* and *postalCode*.
3. Property *isAlive* must be true.
4. Age of contact should be between 18 and 60 years old.

To validate the JSON stored contacts against those requirements, the contacts data file gets converted into the JSON-OBJ format of Algorithm 5 and then it gets tested against the requirements of the company. For this test case, the contacts in Algorithm 4 will be used.

In order to validate the contacts against those requirements, we will need to create an operator that formalizes those requirements and then tries to validate a given JSON-OBJ file against them. This operator, called *properschema* returns true only if all requirements are held. Algorithm 8 contains the operator's signature and equation; given an *ElementList E*, it checks it against all 4 of the requirements. Operator *properschema*:

1. Verifies that the three entries of the first requirement exist. To achieve that, operator *jNodeExists* is called for each entry key name and responds with a boolean.
2. Verifies, using *jNodeExists* operator, whether the three address properties (street address, city and postal code) exist and for each one of them, using *getParent*, checks whether they are child nodes of *address* element.
3. Gets the value of *isAlive* node, using operators *returnJnode*) and *getValue* in succession.
4. Gets the value of *age* node, using operators *returnJnode*) and *getNumber* in succession. The result is checked against the range of allowed values.

In order to verify that the sample JSON-OBJ file (denoted as *samplejson*) holds all desired properties, *properschema* operator has to parse the file. To do that, the *reduce* command

is used; it uses the related operators' equations to try to rewrite the term, eventually returning a boolean value. Output Sequence 1 shows the result of the execution. CafeOBJ returns *true* which means that the JSON data satisfy the four wanted requisites. In case the output was *false*, we can identify through CafeOBJ's output which of the requirements failed to evaluate to true and then act accordingly.

Operator *properschema* acts as a schema /specification in this scenario since it lays out the requirements that any contact should hold and it is also capable of validating any contact against these requirements.

**Algorithm 8:** *properschema* operator

```
op properschema : ElementList -> Bool .
var ElemList : ElementList .
eq properschema(ElemList) =
    if
    -- 1st requirement:
    jNodeExists("Name", ElemList)
    and jNodeExists("address", ElemList)
    and jNodeExists("isAlive", ElemList)
    -- 2nd requirement:
    and (
        jNodeExists("streetAddress", ElemList)
            and getParent("streetAddress", ElemList)
             == "address"
        and jNodeExists("city", ElemList) and
            getParent("city", ElemList) == "address"
        and jNodeExists("postalCode", ElemList) and

            getParent("postalCode", ElemList) ==
             "address" )
    )
    -- 3rd requirement:
    and getValue(returnJnode("isAlive",ElemList))
     == true
    -- 4th requirement:
    and (
        getNumber(returnJnode("age",ElemList)) >=
         18
        and getNumber(returnJnode("age",ElemList))
         <= 60
    )
    then true
    else false
    fi .
```

**Output Sequence 1:** CafeOBJ's reduce command and output

```
-- defining module SCHEMA done.
open SCHEMA .
-- opening module SCHEMA done.
red properschema(samplejson) .
-- reduce in (true):Bool
(0.0000 sec for parse, 0.0640 sec for 502 rewrites +
 10912 matches)
```

## 4. Conclusions and future work

This paper presented an algebraic specification / schema for the JSON Open Standard written in the CafeOBJ language. The proposed formal specification provides a framework for treating JSON objects in an algebraic way that allows formal requirement specification and JSON document validation against a schema.

Formal specifications are in general small in size and our test case is no exception; at only 26 lines of code, it is quite small in size, even with comments present. The requirements in the test case's formal specification are laid out in a clear and non-ambiguous way, provided one can familiarize themselves with the operators and the syntax of the specification. That aside, we believe that such a specification can be quite easy for someone to understand. The way the requirements are presented makes it easier to separate the requirements from one another allowing to track the changes one of them may bring to the whole specification thus reducing any issues of requirement intermixture or confusion. Finally, since this specification is in algebraic format, it is also executable allowing us to verify sample data against the specification.

Despite those advantages, there have not been many attempts in formalizing standards. One reason for this discrepancy could be the plethora of different formal methods that can make early steps such as choosing a tool quite difficult, and also the questionable applicability of a specific formal method to the standard.

The authors hope to continue work on using formal methods to create specifications of more open standards, especially web data standards (e.g. YAML or TOML) in order to achieve benefits over natural language specifications.

**Conflict of Interest** The authors declare no conflict of interest.

## References

[1] G. Blake, R. W. Bly, *The elements of technical writing*, Elements of Series, Longman, 1993.

[2] "Digital imaging and communications in medicine (DICOM)", https://www.dicomstandard.org/current, 2014, accessed: 2022-04-15.

[3] I. Sommerville, *Software Engineering*, Addison-Wesley, Harlow, England, 9 ed., 2010.

[4] K. Barlas, P. Stefaneas, "An algebraic schema for json", "24th Pan-Hellenic Conference on Informatics", PCI 2020, p. 31–33, Association for Computing Machinery, New York, NY, USA, 2020, doi: 10.1145/3437120.3437268.

[5] K. Barlas, E. Berki, P. Stefaneas, G. Koletsos, "Towards formal open standards: Formalizing a standard's requirements", *Innov. Syst. Softw. Eng.*, vol. 13, no. 1, p. 51–66, 2017, doi:10.1007/s11334-016-0283-9.

[6] M. Muhonen, E. Berki, "An open process for quality assurance in systems development", R. Dawson, M. Ross, G. Staples, eds., "Proceedings of Software Quality Management XIX. Global Quality Issues, Loughborough 18-19 Apr., UK", pp. 231–241, 2011.

[7] M. Merruko, "Utilising open source software development for effective electronic health records development", Master's thesis, School of Information Sciences, University of Tampere, 2013.

[8] N. S. Hoe, *Free/Open Source Software, Open Standards*, Elsevier, 2006.

[9] R. Shah, J. Kesan, A. Kennis, "Lessons for open standard policies: a case study of the massachusetts experience", "Proceedings of the 1st international conference on Theory and practice of electronic governance", ICEGOV '07, pp. 141–150, ACM, New York, NY, USA, 2007, doi:10.1145/1328057.1328088.

[10] M. Karjalainen, "Large-scale migration to an open source office suite: An innovation adoption study in finland", phdthesis, University of Tampere, Tampere, 2010.

[11] M. Merruko, E. Berki, P. Nykänen, "Open source software process: A potential catalyst for major changes in electronic health record systems", A. Cerone, D. Persico, S. Fernandes, A. Garcia-Perez, P. Katsaros, S. A. Shaikh, I. Stamelos, eds., "Revised Selected Papers of the SEFM 2012 Satellite Events on Information Technology and Open Source: Applications for Education, Innovation, and Sustainability", vol. 7991, Springer-VerlagBerlin, Heidelberg, 2012.

[12] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, M. Deardeuff, "How amazon web services uses formal methods", *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015, doi:10.1145/2699417.

[13] J. Valtanen, E. Berki, K. Barlas, L. Li, M. Merruko, "Problem-focused education and feedback mechanisms for re-designing a course on open source and software quality.", U. J., B. S., R. M., S. G., eds., "The 18th INSPIRE - INternational conference on Software Process Improvement in Research, Education and Training.", pp. 23–36, London, UK, Southampton Solent University Press., 2013.

[14] "CafeOBJ official site", https://cafeobj.org/, accessed: 2022-04-15.

[15] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, *Introducing OBJ*, pp. 3–167, Springer US, Boston, MA, 2000, doi:10.1007/978-1-4757-6541-0_1.

[16] R. Diaconescu, K. Futatsugi, *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, vol. 6 of *AMAST Series in Computing*, World Scientific Publishing Company, 1998, doi:10.1142/3831.

[17] J. A. Goguen, J. Meseguer, "Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations", *Theoretical Computer Science*, vol. 105, no. 2, pp. 217–273, 1992, doi:10.1016/0304-3975(92)90302-V.

[18] R. Diaconescu, K. Futatsugi, "Behavioural coherence in object-oriented algebraic specification", *Journal of Universal Computer Science*, vol. 6, no. 1, pp. 74–96, 2000, http://www.jucs.org/jucs_6_1/behavioural_coherence_in_object.

[19] T. Sawada, K. Futatsugi, N. Preining, *CafeOBJ Manual*, https://cafeobj.org/files/reference-manual.pdf, ver.1.5.7 ed., 2018.

[20] J. Friesen, *Java XML and JSON: Document Processing for Java SE*, Apress, Berkeley, CA, 2nd ed., 2019, doi:10.1007/978-1-4842-4330-5.

[21] T. Bray, "The javascript object notation (json) data interchange format", RFC 8259, 2017, doi:10.17487/RFC8259.

[22] "Introducing JSON", https://www.json.org/json-en.html, accessed: 2020-05-24.

[23] ECMA, "Standard ecma-404: The json data interchange syntax (1st edition)", *ECMA (European Association for Standardizing Information and Communication Systems)*, 2013.

[24] ECMA, "Standard ecma-404: The json data interchange syntax (2nd edition)", *ECMA (European Association for Standardizing Information and Communication Systems)*, 2017.

[25] M. Droettboom, "Understanding JSON Schema", https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf, accessed: 2022-04-15.

[26] S. C. Reghizzi, L. Breveglieri, A. Morzenti, *Formal Languages and Compilation*, Springer, London, 2019, doi:https://doi.org/10.1007/978-1-84882-050-0.

[27] M. Murata, D. Lee, M. Mani, K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory", *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 4, pp. 660–704, 2005, doi:10.1145/1111627.1111631.



**Dr. KONSTANTINOS BARLAS** has received his bachelor's and integrated master's degree in Applied Mathematical and Physical Sciences from National Technical University of Athens, Greece in 2006. He has completed his PhD degree on Algebraic Methods at the School of Electrical and Computer Engineering of the National Technical University of Athens, Greece in 2018.

His current research includes Algebraic Specifications, Formal Methods, Open Standards, Formal Verification, Formal Semantics and Formal Logic.

**Prof. PETROS STEFANEAS** coordinates the Logic and Formal Methods Group ($\lambda$-ForM) of the Algorithmic Applications and Logic Laboratory at the Department of Mathematics of the National Technical University of Athens, Greece. Dr. Stefaneas has done extensive work on the applications of formal verification and specification techniques to engineering problems.

His current research includes formal methodologies for information privacy, legal documents and open data policies. Other research interests include abstract model theory, algebraic specifications, computational creativity and semantics and ethics of computer science.