PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

FORMAL SPECIFICATION, EXPRESSIVENESS AND COMPLEXITY ANALYSIS FOR JSON SCHEMA

FERNANDO SUÁREZ BARRÍA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JUAN L. REUTTER.

Santiago de Chile, September 2016

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

FORMAL SPECIFICATION, EXPRESSIVENESS AND COMPLEXITY ANALYSIS FOR JSON SCHEMA

FERNANDO SUÁREZ BARRÍA

Members of the Committee:
JUAN L. REUTTER.
CRISTIAN RIVEROS J.
AIDAN HOGAN.
RODRIGO PASCUAL J.

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, September 2016

To my grandparents.

ACKNOWLEDGEMENTS

First of all I would like to thank my parents Vilma and Rodrigo for always being there, since elementary school until the end of my Master's period. Without their personal (and financial) support the result of this work would not have been possible.

I also want to give my gratitude to my advisor Juan, for placing his trust in me and for giving me the opportunity to be his coworker in this process. His ideas and advices were crucial for me in both research and personal aspects. Additionally, I would like to thank Cristian for his unconditional support and for trusting me with the opportunity to give lectures at university. Besides I would like to thank the following people:

- Martín Ugarte for giving me his help and mentoring every time I asked for it.
- Domagoj for all his help and ideas for my research problems.
- Adrián for always providing me moral support during my office hours.

Finally I would like to thank Catalina as well as all my college and high school friends for the good times, and for taking my college experience to a whole new level.

My postgraduate studies were partially funded by The Millennium Nucleus Center for Semantic Web Research under Grant NC120004, and Fondecyt Iniciación grant 11130648.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

JSON – the most popular data format for sending API requests and responses – is still lacking a standardised schema or meta-data definition that allows developers to specify the structure of JSON documents. JSON Schema is an attempt to provide a general purpose schema language for JSON, but it is still work in progress, and the formal specification has not yet been agreed upon. Why this could be a problem becomes evident when examining the behaviour of numerous tools for validating JSON documents against this initial schema proposal: although they agree on most general cases, when presented with the greyer areas of the specification they tend to differ significantly.

In this thesis we conduct a first theoretical analysis for JSON Schema documents. We start by providing the first formal definition of syntax and semantics for the whole JSON Schema specification. Then, we use our definition to capture the expressive power of JSON Schema in terms of two well established formalisms, automata theory and logic. First, we show that even when using a very restricted set of keywords, we can simulate nondeterministic tree automata. On the other hand, we also show that monadic second order logic captures a core fragment of the JSON Schema specification.

Finally, we attempt to establish tight complexity bounds for the most important questions in the context of schema definitions, the validation and satisfiability problems. These problems are crucial for the development of efficient algorithms for data transfer over the web. Here we show that the validation problem can be efficiently solved when the schema is fixed, but inherently sequential in terms of combined complexity. On the other side, we provide an exponential algorithm for solving satisfiability and also show that it cannot be sorted out in a better upper bound.

Keywords: JSON, API, JSON Schema, Syntax, Semantics, Complexity, Expressive power, Tree automata, Monadic second order logic

RESUMEN

JSON es hoy en día el formato más popular para el traspaso de datos en la web. Sin embargo, todavía carece de un esquema estandarizado que permita a desarrolladores especificar la estructura estos documentos. JSON Schema es una propuesta para definir *metadata* a través de esquemas para documentos JSON. Sin embargo, todavía está en una etapa de madurez muy temprana, y no existe una especificación formal que capture al lenguaje en su totalidad. En esta tesis se lleva a cabo el primer análisis formal de documentos JSON Schema.

En primer lugar, se propone una definición formal tanto de la sintaxis como de la semántica para la validación de estos documentos. Luego, en base a esta definición, se procede a capturar el poder expresivo del lenguaje. Por un lado, se demuestra que JSON Schema puede simular autómatas de árbol, con el sólo uso de un subconjunto muy restringido de la especificación. Por otro lado, se muestra que JSON Schema puede ser capturado por lógica monádica de segundo orden sobre árboles.

Finalmente, se analiza la complejidad computacional de los principales problemas de decisión presentes en el contexto esquemas para lenguajes formales, el problema de validación y el problema de satisfacibilidad para documentos JSON Schema. Estos problemas resultan de gran importancia en el desarrollo de algoritmos eficientes para el traspaso de datos en la *web*. Por un lado, se muestra que el problema de validación puede ser resuelto de manera eficiente para esquemas fijos. Sin embargo, en términos de complejidad combinada el problema resulta ser inherentemente secuencial. Por otro lado, se propone un algoritmo exponencial para resolver el problema de satisfacibilidad. Además, se demuestra que este problema no puede ser computado con una mejor cota que la obtenida.

Palabras Claves: JSON, JSON Schema, Complejidad computacional, Validación, Satisfacibilidad, Poder expresivo, Autómata, Lógica

1. INTRODUCTION

1.1. Background

JSON (JavaScript Object Notation) (Bray, 2014; Internet Engineering Task Force (IETF), 2014) is a structured data format based on the data types of the JavaScript programming language. In the last few years JSON has gained tremendous popularity among web developers, and has become the main format for exchanging information between servers over the web.

Nowadays, JSON plays a key role in web applications. Indeed, software executing functions ordered by remote machines must establish a precise protocol for receiving and answering requests, a problem that for all intents and purposes, has been solved by Application Programming Interfaces (API). Given that JSON is a language which can be easily understood by both developers and machines, it has become the most popular format for sending API requests and responses over the HTTP protocol. As an example, consider an application containing information about the estimated time of arrival for public transportation. The application provides an API to allow other software to access this information. A hypothetical call to this API could be a request containing this JSON file:

```
{"street_name": "Broadway", "street_number": 4304}
```

FIGURE 1.1. Example of an API input in JSON format

by which a client is requesting the estimated time of arrival of a bus in the address Broadway 4304. Given this call, the API would reply with an HTTP response containing the following JSON file:

```
{"bus_code": "C01",
 "minutes_before_arrival": 15,
 "seconds_before_arrival": 40}
```

FIGURE 1.2. Example of an API output in JSON format

indicating that the bus C01 is going to arrive in 15 minutes and 40 seconds. This example illustrates the simplicity and readability of JSON, which partially explains its fast adoption.

With the popularity of JSON, it was soon noted that in many scenarios one can benefit from a declarative way of specifying a schema for JSON documents.

For instance, in the public API scenario one could use a schema to avoid receiving malformed API calls that may affect the inner engine of the application. Coming back to the public transport application, note that the API calls consist of JSON objects mentioning a string (the street name) and a number (the street number) . What happens if a user does not specify one of these properties, or if he or she specifies more properties in the JSON object? Similar issues arise when we use a number or a boolean value instead of a string or vice versa. Without an integrity layer, all of these questions need to be taken into consideration when coding the API, and could be avoided if we use a schema definition to filter out documents that are not of the correct form. A declarative schema specification would also give developers a standardised language to specify what types of JSON documents are accepted as inputs and outputs by their API.

JSON Schema (*json-schema.org: The home of JSON schema*, 2016) is a simple schema language that allows users to constrain the structure of JSON documents and provides a framework for verifying the integrity of the requests and their compliance to the API. If we consider again the public transport API, by simply adding the JSON Schema documento from Firure 1.3 we can assure the correct form of each API call.

This schema asserts that the received JSON document must be of type object (a collection of key-value pairs), besides, it must contain keys “street_name” and “street_number”, and there cannot be any more keys. Moreover, the schema forces the“street_name” to be a string and “street_number” to be an integer. For example, the JSON file requesting the estimated time of arrival in Broadway 4304 would comply to this schema, but the JSON file

```
{
  "type": "object",
  "required": ["street_name", "street_number"],
  "additionalProperties": false,
  "properties": {
   "street_name": {"type": "string"}
   "street_number": {"type": "integer"}
  }
}
```

FIGURE 1.3. Example of a JSON Schema document

`{"street_name": "Wall st", "street_number": "fifty three"}` would not, as the value of the street number is not an integer.

To the best of our knowledge, JSON Schema is the only general attempt to define a schema language for JSON documents, and it is slowly being established as the default schema specification for JSON. The definition is not yet a standard (the specification is currently in its fourth draft (Galiegue & Zyp, 2013)), but there is already a growing body of applications that support JSON schema definitions, and a great amount of tools and packages that enable the validation of documents against JSON Schema. There have been other alternatives for defining schemas for JSON documents, but these are either based on JSON Schema itself or have been designed with a particular set of use cases in mind. To name a few of them, Orderly (Hilaiel, 2015) is an attempt to improve the readability of a subset of JSON Schema, Swagger (*Swagger: The World's Most Popular Framework for APIs.*, 2015), RAML (The RAML Workgroup, 2015) and Google discovery (Google, 2015) are proposals for standardising API definition that use JSON Schema, and JSON-LD (Sporny, Kellogg, & Lanthaler, 2014) is a context specific definition to specify RDF as JSON.

Despite all the advantages of a schema definition, the adoption of JSON Schema has been rather slow and there are not many studies about this topic. To the best of our knowledge, the only rigorous analysis related with JSON Schema is the one done by Reutter, Ugarte, and Vrgoc (2015), but it was quite general and brief. In this context, one of the

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

The lack of a formal definition has also discouraged the scientific community to get involved: to the best of our knowledge, there has been no formal study of general schema specifications for JSON, nor has there been any formal discussion regarding the design choices taken by the JSON Schema specification. A formal specification would also help the development of automation tools for APIs. There is already software for automatically generating documentation (GitHub, 2013b) and API clients (GitHub, 2013a, 2013c), but all of them suffer from the same problems as validators.

Another consequence of this problem, is that there is no real knowledge about the expressiveness nor the computational complexity for these schema definitions for JSON documents. A formal analysis could be beneficial in terms of delimiting the real potential of the language in the context of data storage and transfer. For example, there is no real knowledge about how hard it is to validate a JSON document against a JSON Schema, in terms of time and memory. This is a crucial point, given the growing amount of requests APIs receive each day. For example, Twitter's API receives more than three billion request each day, all of them in JSON format (Rao, n.d.). Moreover, with the passage of time, these APIs get more complex in terms of features and diversity of inputs and/or outputs. In this context, a formal analysis would serve as the starting point for developing efficient algorithms for generating automatic documentation of these services. Furthermore, an expressiveness analysis could serve to finally devise the main differences between JSON and its most representative comparisson: XML.

In this thesis, we provide both a formal specification and a theoretical analysis for the most significant problems related with JSON Schema documents. Our specification

include both a clear syntax and semantics for each of the features of the current specification. As mentioned, this is very important, since it can serve as a basis to standardize the construction of JSON Schema documents and the way they are validated. Afterwards, we use our specification to conduct a theoretical analysis of JSON Schema documents. We begin by delimiting the expressive power of JSON Schema, in comparison to *tree automata* and *monadic second order logic*. Then, we conduct a computational complexity study for the main problems in the context of schema languages. We start by analysing the computational cost of verifying if a JSON document conforms to a schema, also known as the *validation problem*. As mentioned before, this problem has a key role in the adoption of JSON Schema as a standard for specifying the protocol we use to communicate with web services. Finally, we analyse the problem of checking the existence of documents that comply to a given schema, a problem we call the *satisfiability problem*. This problem holds a strong relation with the problems related with the documentation of APIs. For example, in order to design an algorithm to provide a fixed amount of examples for the input of an API (given the schema), it is necessary to solve the satisfiability problem before. Moreover, the problem is also necessary to provide algorithms to automate the construction of schemas given a set of JSON documents as input, a problem that is very interesting in the topics of machine learning. Furthermore, it can be useful for optimizing document-oriented database management systems. For example, one could check if the restrictions above certain database are consistent before the insertion of tuples into the dataset.

1.2. Summary of contributions

In the first chapter, we propose a solution for the lack of a formal specification of JSON Schema. Specifically, we provide a context-free grammar that captures the whole syntax of JSON Schema documents. Moreover, we give a formal definition of *well-formed schemas* and how they are constructed. Here we also give a full definition of the semantics for each restriction present in the current specification.

In chapters 2–3, we conduct a formal study of several aspects of the JSON Schema specification. We start by studying the expressive power of JSON Schema as a language for defining classes of JSON documents. Since JSON Schema is the only native schema definition for JSON we cannot compare to other standards; instead we provide comparison with respect to automata models and logic, the two most important theoretical yardsticks for expressive power. Here we prove the following results:

- JSON Schema can simulate non deterministic tree automata on ranked trees, even if we allow a very restricted set of keywords.
- The family of non-array schemas is captured by monadic second order logic.

We continue with a complexity analysis on the main problems related with JSON Schema. First, we study the problem of validating a JSON document against a schema, providing tight bounds for the computational complexity of this problem. Here we have the following results:

- The validation problem is PTIME-complete in combined complexity for the whole class of JSON Schema documents.
- If we we don't allow array restrictions, the validation problem can be computed in linear time in terms of data complexity.

Finally we analyse the problem of checking whether or not there exists a document that conforms to a fixed schema, also known as the satisfiability problem. Regarding this problem, we have the following results in combined complexity:

- The satisfiability problem is EXPTIME-complete for the whole family of JSON Schema documents.
- The satisfiability problem is PSPACE-complete for non-recursive schemas.
- The satisfiability problem is NP-complete for non-recursive non-regular schemas.

1.3. Thesis outline and structure

Chapter 2 shows the problems we run into because of the lack of a formal specification, and defines the syntax and semantics of JSON Schema. In Chapter 3 we give a formal analysis of the expressive power of JSON Schema. Next, in Chapter 4 we provide a full complexity analysis for the validation and satisfiability problems. Finally, Chapter 5 presents concluding remarks and some possible future lines of research.

2. DEVISING A FORMAL SPECIFICATION FOR JSON SCHEMA

As we mentioned in the previous section, one of the main problems of JSON Schema is the lack of a formal specification. To illustrate why this is an issue we created four border-case schemas, and validated them using five different validators. These tests use schemas that are *allowed by the current JSON Schema draft* (Galiegue & Zyp, 2013), but the valuation of their features is not fully specified by the draft. The first test (T1) evaluates whether or not a collection of key-value pairs is considered to be ordered. The second test (T2) checks the behaviour of validators for a schema specifying both that the document is an integer and a string. Next, the test (T3) states that the document is an object, but also adds an integer constraint to it. Lastly, (T4) uses definitions and references to force an infinite loop, while also allowing the object to be a simple string. For space reasons the full code of the tests and their details are left out from the body of this thesis, but can be found in Appendix A.1 and A.2 .

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

Y	valid
N	invalid
–	unsupported

TABLE 2.1. Testing five validators against 4 tests.

Table 2.1 shows the outcome of this process. It is important to mention that all validators successfully validate the JSON Schema test-suite (Berman, 2015). As we can see, no two validators behave the same on all inputs, which is clearly not the desired behaviour. This illustrates the need for a formal definition of JSON Schema which will either disallow ambiguous schemas, or formally specify how these should be evaluated.

2.1. Notation

We start by fixing some notation regarding JSON documents and introducing JSON Pointer, a simple query language for JSON that is heavily used in the JSON Schema specification. For readability we skip most of the encoding details with respect to these specifications; their formal definition can be found in (Bray, 2014; Internet Engineering Task Force (IETF), 2013).

2.1.1. JSON Values

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

We sometimes use the term JSON document (or just document) to refer to JSON values. The following syntax is normally used to navigate through JSON documents. If J is an object, then $J[\text{“key”}]$ is the value of J whose key is the string “key”. In the case that “key” is not in J, we denote $J[\text{“key”}] = \emptyset$. Likewise, if J is an array, then $J[n]$, for a natural number n, contains the $(n-1)$-th element of J. Similarly as for objects, in the case that an element is not present we say that $J[n] = \emptyset$.

2.1.2. JSON Pointer

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

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

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

Example 1. Consider now an array storing names

```
J = [{"name":  "Josefine"}, {"name":  "Michael"}]
```

To extract the value of the key `"name"` for the second object in the array, we can use the JSON pointer `p = /1/name` which first navigates to the second item of the array (thus obtaining the object {`"name":  "Michael"`}) and retrieves the value of the key `"name"` from here. Therefore Eval$(p, J) =$ `"Michael"`.

2.2. Formal Grammar

JSON Schema can specify any of the six types of valid JSON documents: objects, arrays, strings, numbers, boolean values and null; and for each of these types there are several keywords that help shaping and restricting the set of documents that a schema specifies. For the sake of space, we have identified a core fragment that is equivalent to the full JSON Schema specification, and present now its formal grammar and semantics. All of the remaining functionalities in the official JSON Schema draft can be expressed using the functionalities included in this section. The complete grammar definition can be found in Appendix B.1 . It is important to note that this work was done in conjunction

with the community (*json-schema.org: The home of JSON schema*, 2016; Galiegue & Zyp, 2013).

The compacted grammar is presented in Tables 2.2–2.6. It is specified in a visual-based extended Backus-Naur form (The International Organization for Standardization (ISO), 1996), where all non-terminals are written in bold (and thus everything not in bold is a terminal). Also, for readability, we use **string** to represent any JSON string, **n** to represent any positive integer, **r** to represent any decimal number, **Jval** to represent any possible JSON document and **regExp** to represent any regular expression. Note that when these values get instantiated they behave as terminals.

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

2.2.1. Overall Structure

Table 2.2 defines the overall structure of JSON Schema document (**JSDoc**). It consists of two parts: an optional *definitions* section (**defs**), that is intended to store other schema definitions to be reused later on, and a mandatory *schema* section (**JSch**) where the actual schema is specified. In turn, each schema can be either a *string schema* (**strSch**), a *number schema* (**numSch**), an *integer schema* (**intSch**), an *object schema* (**objSch**), an *array schema* (**arrSch**), a *reference schema* (**refSch**), a boolean combination of schemas using **not**, **allOf** or **anyOf**, or simply the *enumeration* of a set of values (**enum**). Note how reference schemas make use of JSON pointer (**JPointer**).

2.2.2. Strings

String schemas are formed according to Table 2.3. We first state that we wish to represent a string using the `"type":"string"` pair, and then we may add an additional

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

TABLE 2.2. Grammar for JSON Schema documents

restrictions to state that they satisfy a certain regular expression **regExp**. We illustrate some of these concepts in Example 2.

strSch	:=	`"type"` **:** `"string"` (, **strRes**)*
strRes	:=	**pattern**
pattern	:=	`"pattern"` **:** **"regExp"**

TABLE 2.3. Grammar for string schemas

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

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

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

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

```
        "not": {"$ref": "#/definitions/email"}
    }
```

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

2.2.3. Numeric Values

Integer and number schemas have the same structure, shown in Table 2.4. The pair `"type":"number"` specifies any number, while `"type":"integer"` specifies integers only. We can specify maximum and/or minimum values for numbers and integers (these values are not exclusive unless explicitly stated), and also that numbers and integers should be multiples of another number.

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

TABLE 2.4. Grammar for numeric schemas

2.2.4. Objects

We specify object schemas with the `"type":"object"` pair, and according to Table 2.5. Within objects schemas we may use additional restrictions to control the key-value pairs inside objects. The keyword `required` specifies that a certain string needs to be a key of one of the pairs inside an object, and `properties` is used to state that the value of a key needs itself to satisfy a certain schema. The keyword `patternProperties`

JSON Schema treats integers as a different type.

works like `properties`, except we bound all key-value pairs whose key satisfies a regular expression, and finally `additionalProperties` controls whether we allow any additional key-value pair not defined in `properties` or `patternProperties`.

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

TABLE 2.5. Grammar for object schemas

Example 3. Recall the schema describing an API call to the public transport app defined in Figure 1.3. As the API is expecting a JSON containing a street name and a street number, but nothing else. Moreover, our schema specifies that the street name must be a string and the street number must be an integer. We also use *`required`* and *`additionalProperties`* to specify that the JSON we are sending to the app will contain precisely those two keys and nothing else.

2.2.5. Arrays

Finally, array schemas are specified with the `"type":"array"` pair, and according to Table 2.6. There are two ways of specifying what kind of documents we find in arrays. If a single schema follows the `"items"` keyword, then every document in the array needs to satisfy this schema. On the other hand, if an array follows the `"items"` keyword, then it is one-by-one: the i-th document in the specified array needs to satisfy the i-th schema that comes after the `"items"` keyword. We can also set a minimum and/or a maximum

number of items, and finally we can use `uniqueItems` to specify that all documents inside an array need to be different.

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

TABLE 2.6. Grammar for array schemas

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

```
{
  "type": "array",
  "items": {"$ref": "#/definitions/schema_example_3"}
}
```

2.3. Semantics

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

the semantics for the compacted version of the JSON Schema Draft definition. However, the remaining functionalities are defined in Appendix B.2.

2.3.1. Combinations and References

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

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

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

2.3.2. Strings

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

- p is `"pattern":` `e` and J is a string that belongs to the language of the expression `e`.

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

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

and thus $J \not\models S_2$.

2.3.3. Numeric values

Let S be a number (respectively, integer) schema. Then $J \models S$ if J is a number (resp. integer), and for each key-value pair p in S whose key is not `"type"`, `"exclusiveMinimum"` nor `"exclusiveMaximum"` one of the following holds:

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

2.3.4. Objects

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

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

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

  Remark. If the keyword matches more than one pattern property then it has to satisfy all the schemas involved.
- p is `"required":` $[k_1, \ldots, k_m]$ and for each $1 \leq j \leq m$ we have that J has a pair of the form $k_j : v$.
- p is `"additionalProperties":` `false` and for each pair $k : v$ in J, either S contains `"properties":` $\{k_1 : S_1, \ldots, k_m : S_m\}$ and $k = k_j$ for some $1 \leq j \leq m$, or S contains `"patternProperties":` $\{$`"`e_1`":` $S_1, \ldots,$ `"`e_m`":` $S_m\}$ and k belongs to the language of e_j, for some $1 \leq j \leq m$.
- p is `"additionalProperties":`S' and for each key-value pair $k' : j'$ in J, with k' not in S[properties] and k' not matching any of the expressions in S[patternProperties], we have that j' validates against S'

2.3.5. Arrays

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

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

Example 6. Consider the document $J = [7, -3, 2, -4]$ and the schema S as follows

{

```
        "type": "array",
        "uniqueItems": true,
        "items":{"type": "integer"}
    }
```

The schema S is asking for array documents whose elements must be different from each other. Moreover, the schema forces those elements to be integers. It is clear that J comply to these restrictions, then we obtain that $J \models S$.

2.4. Well Formedness

Up to this point, we have defined both a proper syntax and semantics for JSON Schema documents. But before we can conduct a formal analysis for the specification, we have to deal with some border case schemas that may be problematic. For example, the formal grammar still allows some enigmatic schemas, such as the one in Figure 2.1.

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

FIGURE 2.1. Example of an ill-designed schema

The example above defines a schema that is both S and the negation of S, and is therefore ill-designed. What is worse, the majority of the validators we tested run into an infinite loop when trying to resolve the references of this schema. In fact, one of the tests in Table 2.1 used a schema similar to this one.

To avoid these problematic cases we introduce the notion of a *well-formed* schema. Intuitively, what's going on behind the structure of the schema is that we have a non-valid self reference. In order to avoid these references, we can check the "reachability" between

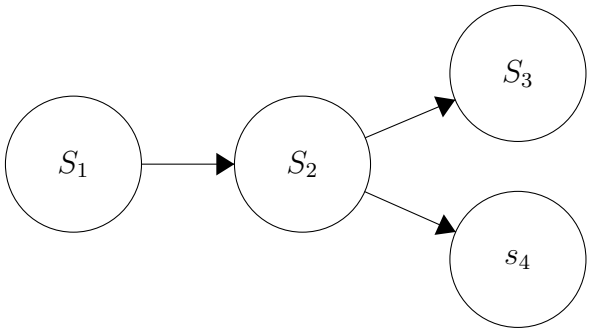
the schemas named under the "definitions" part. This idea motivates the following definition:

Definition 1 (Reference graphs)**.** Given a JSON Schema document S, the *reference graph* of S is a graph that has a node for each schema defined under the "definitions" section of S. Let S_i, S_j be schemas under the definitions section, the reference graph has an edge from S_i to S_j if and only if S_i is a boolean combination of schemas and S_j is one of those schemas.

Example 7. Consider the following JSON Schema document S.

```
{
    "definitions":{
        "S_1":{"$ref":"#/definitions/S_2"},
        "S_2":{
            "allOf":[
                { "$ref":"#/definitions/S_3"},
                { "not":{ "$ref":"#/definitions/S_4"}}
            ]
        },
        "S_3": {"type": "string"},
        "S_4": {"type": "object"}
    },
    "$ref":"#/definitions/S_1"
}
```

Now we can build the following reference graph from S.



Finally, we can use this definition to establish well formedness in JSON Schema documents.

Definition 2 (Well-formed schema)**.** Given a JSON Schema document S, we say that S is well formed if its reference graph is acyclic.

For instance, the graph of Example 7 is a well formed schema, but the one in Figure 2.1 is not, since it has a self loop on S. For the rest of the paper we consider only well formed schemas, and we propose to add this condition to the standard as well. Note also that well formedness can be checked in linear time by performing DFS over the tree (Cormen, Stein, Rivest, & Leiserson, 2001).

3. EXPRESSIVENESS

So far we have seen many examples of how JSON Schema can be used in practice, but we still do not know much about the classes of JSON documents that the JSON Schema specification can express, and which ones it cannot. These questions were also asked when XML Schema was being studied and were solved by checking the expressive power in comparison to established formalisms. For JSON Schema we provide two such comparisons: with respect to automata and with respect to logic.

Remark. For the sake of simplicity, in this part of the work we just focus on well-formed schemas constructed according to the compacted grammar (Section 2.2). However, all the results presented in this section can be adapted to apply to the whole specification.

3.1. Preliminaries

We assume familiarity with tree automata theory (Comon et al., 2007), computational complexity (Papadimitriou, 1994) and logic (Libkin, 2004, 2006). Now we revise some concepts of tree codification and automata theory.

3.1.1. Σ-Trees

We start by defining one of the most common abstractions for tree encoding: Σ-trees. These definitions are the same used in (Neven, 2002) and we need them to code JSON as tree structures. The set of Σ-trees, denoted by $\mathcal{T}_\Sigma$, is inductively defined as follows:

- every $\sigma \in \Sigma$ is a Σ-tree,
- if $\sigma \in \Sigma$ and $t_1, ..., t_n \in \mathcal{T}_\Sigma, n \geq 1$ then $\sigma(t_1, ..., t_n)$ is a Σ-tree.

Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. Denote by $\mathbb{N}^*$, the set of strings over the alphabet consisting of the natural numbers. For every tree $t \in \mathcal{T}_\Sigma$, the set of nodes of t, denoted by Dom(t), is the subset of $\mathbb{N}^*$ defined as follows: if $\sigma(t_1, ..., t_n)$ with $\sigma \in \Sigma, n \geq 0$, and $t_1, ..., t_n \in \mathcal{T}_\Sigma$,

then Dom(t)= $\{\epsilon\} \cup \{ui \mid i \in \{1, ..., n\}, u \in Dom(t_i)\}$. Thus, ϵ represents the root thile ui represents the i-th child of u. This definition is also known as *tree domain* (Libkin, 2006). From now on we will refer to Σ-trees just as trees.

3.1.2. Non-deterministic Top Down Tree Automata

The most common model that we use throughout our analysis is *Non-deterministic Finite Tree Automata* (NFTA). Here we provide the top-down definition from (Comon et al., 2007) that is used in the proofs along this thesis.

A nondeterministic **top-down** finite Tree Automaton (top-down NFTA) over $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, \mathcal{I}, \Delta)$ where Q is a set of states (states are unary symbols), $\mathcal{I} \subseteq Q$ is a set of initial states, and Δ is a set of rewrite rules of the following type:

$$q(f(x_1, ..., x_n)) \rightarrow f(q_1(x_1), ..., q_n(x_n))$$

where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, ..., q_n \in Q, x_1, ..., x_n \in \mathcal{X}$.

When $n = 0$, i.e. when the symbol is a constant symbol a, a transition rule of top-down NFTA is of the form $q(a) \rightarrow a$. A top-down automaton starts at the root and moves downward, associating along the run a state with each subterm inductively. We do not formally define the move relation $\rightarrow_{\mathcal{A}}$ because the definition is easily deduced from the corresponding definition for bottom-up NFTA which can be found at (Comon et al., 2007). The tree language $L(\mathcal{A})$ recognised by $\mathcal{A}$ is the set of all ground terms t for which there is an initial state q in $\mathcal{I}$ such that

$$q(t) \underset{\mathcal{A}}{\overset{*}{\rightarrow}} t$$

3.2. From Automata Theory to JSON Schema

Most schema definitions for other semistructured data paradigms are heavily based on automata. In the case of XML, Martens, Neven, Schwentick, and Bex (2006) linked

schema definitions with different versions of tree automata. It is therefore useful to compare with automata formalisms, so we can understand how much does JSON Schema depart from XML Schema formalisms.

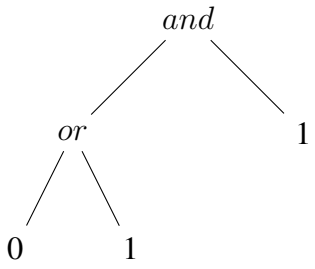
We begin by showing that JSON schema can define any standard non-deterministic finite tree automaton (NFTA). For the proof we just focus on automata for binary trees, since the expressive power of these automata is independent of their arity, as mentioned in (Neven, 2002) . To formally state this, we encode every tree $T \in \mathcal{T}$ into a JSON document $J(T)$. This encoding is inductively defined as follows:

(1) if T is of the form $\sigma(t_1, t_2)$, then $J(T) = \{\sigma : \{\texttt{lChild}:\{J(t_1)\},\ \texttt{rChild}:\{J(t_2)\}\}$

(2) if T is of the form σ then $J(T) = \sigma : \texttt{null}$

To illustrate this mapping, consider the following example:

Example 8. A tree T and its encoding into a JSON document $J(T)$:




```
{ "and": {
   "lChild": {
      "or": {
         "lChild": {"0": null },
         "rChild": {"1": null}}},
   "rChild": {"1": null}
   } }
```

The idea of the proof is to show that for every NFTA $\mathcal{A}$ one can construct a schema $S(\mathcal{A})$ such that a tree T belongs to the language of $\mathcal{A}$ if and only if $J(T) \models S(\mathcal{A})$. To illustrate this claim consider the automaton $\mathcal{A} = (Q, \mathcal{F}, \mathcal{I}, \Delta)$ defined by: $\mathcal{F} = \{and(\cdot,\cdot), or(\cdot,\cdot), 0, 1\}$, $Q = \{q_0, q_1\}, \mathcal{I} = \{q_1\}$ and $\Delta =$

$$
\begin{array}{lll}
\{ & q_1(and) \to (q_1, q_1), & q_0(or) \to (q_0, q_0), \\
& q_1(or) \to (q_1, q_1), & q_0(and) \to (q_0, q_0), \\
& q_1(or) \to (q_1, q_0), & q_0(and) \to (q_1, q_0), \\
& q_1(or) \to (q_0, q_1), & q_0(and) \to (q_0, q_1), \\
& q_1 \to 1, & q_0 \to 0 \qquad \}
\end{array}
$$

The automaton above is intended to accept the tree encoding of positive output circuits. For instance, the tree from example 8 belongs to the language of $\mathcal{A}$. The full transformation is given in Figure A.1 (Appendix A.4). To obtain a schema that accepts (up to our coding) only the trees in the language of $\mathcal{A}$ we proceed as follows. First, in the `"definitions"` section of our schema we define each state of the automaton. Figure A.1 illustrates how is this done for the automaton above. Namely, we have a schema for q_0 and q_1. Each of these schemas is intended to code the transition from the state it describes. This is achieved by declaring that each state is an object whose properties code the transitions leaving the state. For instance, in order to simulate that we can move from the state q_1 to the pair of states (q_0, q_1) reading the symbol or, we add

```
"or": {
        "type": "object",
        "properties": {
          "lChild": {"$ref": "#/definitions/q0"},
          "rChild": {"$ref": "#/definitions/q1" }
        },
        "additionalProperties": false,
        "required": ["lChild", "rChild" ]
        }
```

to the properties of the schema for q_1. Note that here we use `$ref` to switch to the schema of q_1 and follow the transition. Likewise, to reflect that a non deterministic choice is available, we use the `anyOf` keyword. For instance, this is reflected in Figure A.1 when describing the transitions of q_1. Additionally, in order to signal that a state is accepting, we allow it to be of type `null`, such as e.g. for q_0 and q_1. Finally, we add an `anyOf` keyword in the body of the schema, containing reference to all the initial states (e.g. q_1).

It is now straightforward to see that a tree T belongs to the language of the automaton $\mathcal{A}$ if and only if $J(T)$ validates against the schema from Figure A.1. For example, if we take the tree T from example 8, it is easy to check both $T \in L(\mathcal{A})$ and $J(T) \models S$, where S is the schema from Figure A.1.

Although the procedure described above gives a mapping from one particular automaton in linear time, we provide the complete construction in Appendix C.1. We therefore obtain the following:

PROPOSITION 1. *JSON Schema can simulate non deterministic tree automata even when it only uses definitions, references, single enumeration and boolean combinations of schemas.*

The result above is important for several reasons. First of all, it shows us that even when using a very restricted set of keywords, we can simulate tree automata. This means that the core operators of JSON Schema already allow the representation of a large class of problems.

The other consequence, as we mentioned in the introduction, is that we now know with certainty that the logic behind JSON Schema is, at least, comparable to that of XSDs and other XML Schema specifications. This follows from the fact that the logic of these formalisms is captured by tree automata, as shown in (Martens et al., 2006).

But is the converse true? Can we use automata to somehow simulate the validation process of JSON Schema? Unfortunately, this is not the case; consider the schema presented in Figure 3.1.

This schema defines a nesting of arrays where each element is itself a JSON document that conforms to `S`: either the `null` value or an array with exactly two elements that again must conform to `S`. We can look at these objects as *complete binary trees*, where arrays represent inner nodes and `null` elements represent leafs. For example, a tree representation of a JSON document that conforms to this schema is presented in Figure 3.2.

It is easy to note that the conflictive restriction here is the `uniqueItems` keyword, since when it is negated it could force child nodes to have the same value. As consequence, just with this schema we can simulate the whole family of complete binary trees, a language that cannot be captured by NFTA (Comon et al., 2007).

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

FIGURE 3.1. JSON Schema document that captures complete binary trees.

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

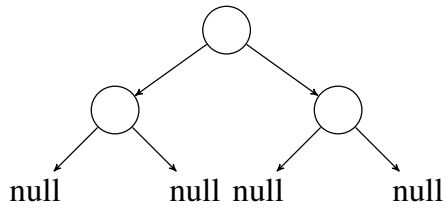


FIGURE 3.2. Binary tree coded as a JSON document.

3.3. From JSON Schema to MSO

When studying basic properties of formal specifications such as the one of JSON Schema, one is often interested in whether they can be expressed in well studied formalisms such as first or second order logic. Not only will this allow us to apply known results on the expressiveness of our specification, but it can also give us some insight into how difficult certain problems are computationally. As we know that JSON Schema documents can simulate finite state automata, we next show that these documents can be expressed in monadic second order logic (MSO), a powerful logical formalism often used to cover languages containing some form of regular expressions (Libkin, 2004). For ease of exposition, we would like to avoid to have tree structures with an ordered partition and an unordered one. Because of this and given the fact that ordered trees have been already well studied (classic tree automata runs over ordered trees), we just focus on non-array

schemas. However, one can show that our results continue to hold for any fragment of the full JSON Schema specification (*json-schema.org: The home of JSON schema*, 2016) that does not use the `uniqueItems` keyword (which we show not to be definable in MSO).

To understand the connection with logic it is best to consider every JSON instance J as an unranked unordered tree $T(J)$ (recall that we do not consider arrays here), whose leafs are either empty object instances or strings. For example, Figure 3.3 shows a simple JSON document and its representation as a tree structure.

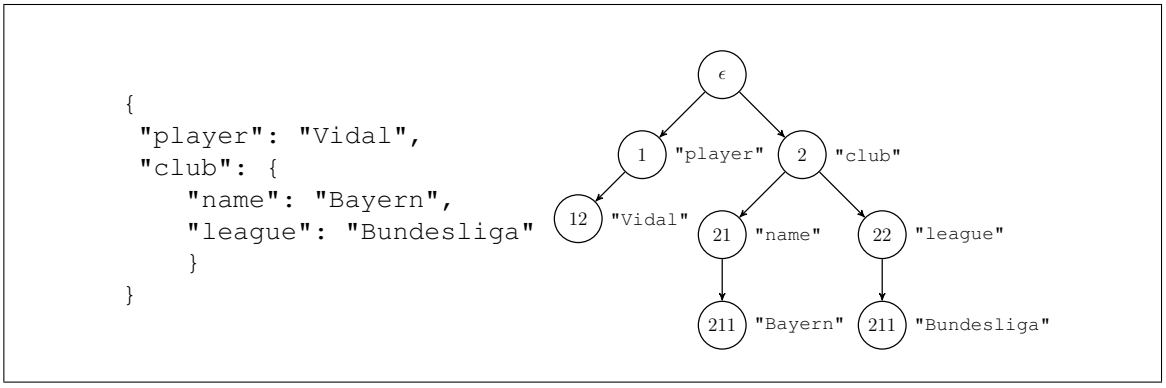


FIGURE 3.3. A JSON J and its tree representation $T(J)$.

The structures that we use to code trees conform to the following scheme:

$$\mathfrak{T} = \langle T_D, \Sigma^*, \prec_{ch}, P_{root}, \lambda, (R_L)_{L \subseteq \Sigma^*} \rangle$$

where $T_D \subseteq \mathbb{N}^*$ is an appropriate tree domain such as the one described in section 3.1 together with the set of all strings appearing in the JSON document Σ^*. Besides, the *child relation* is defined over $\mathbb{N}^*$ as follows:

$$v \prec_{ch} v' \text{ if and only if } s' = s \cdot i \text{ for some } i \in \mathbb{N}$$

We also use the root relation P_{root} over $\mathbb{N}^*$ just for the convenience of placing a label at the root of the document. Additionally, we need the *labeling* mapping $\lambda : T_D \to \Sigma^*$ to assign either a keyword or a string value to each node. Finally, we have a predicate for each possible regular expression $(R_L)_{L \subseteq \Sigma^*}$; this is intended to model the pattern restrictions

Note that the empty object $\{\}$ does not have a string assigned by λ (i.e it is a partial function).

we have in the specification. This is done by constructing a formula for each regular expression such that the formula is satisfied by a string if and only if the string belongs to the language of the expression. In this context, we should mention that even if we are working over a tree domain, we could easily adapt these structures to support another sorting that simulate these predicates. Regardless of this, we use the predicate notation to keep the model clean and understandable. Note that we have the faculty to do this since we are working around monadic second order logic.

The key observation is that each schema can be described using an MSO formula over $T(J)$. Let us start with an example, consider the schema S below.

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

This schema specifies all objects that have a player attribute whose value is a string. In particular, the document from Figure 3.3 validates against this schema. An MSO formula equivalent to this schema would be:

$$\exists x \forall y (P_{root}(x) \wedge (x \prec_{ch} y \wedge \lambda(y) = \texttt{"player"} \rightarrow \exists z \exists s \forall v (y \prec_{ch} z \wedge \lambda(z) = s \wedge z \not\prec_{ch} v))$$

Intuitively, this formula checks that every time that the root has a child labeled "player", that child must have another child labeled with a string. Using the codification from Figure 3.3 this is equivalent to saying that the JSON document could have a key-value pair with the key being `"player"` and in that case, the value must be a string, as desired. Other JSON Schema constructs can be simulated by MSO operators in a similar way. Note that we need to use second order properties only to deal with definitions and references, and to simulate regular expressions; all other keywords are expressible in first order logic.

Generalising this construction in a similar way as done in (Libkin, 2006), we obtain the following.

THEOREM 1. *For any non-array schema S there exists an MSO formula F_S such that for every JSON document J we have that $J \models S$ if and only if $T(J) \models F_S$.*

The proof comes by encoding each possible JSON Schema document into an MSO formula, the full demonstration for non-array schemas can be found at Appendix C.2. Regardless of this, we conjecture that it is possible to extend this result to apply for every JSON schema not using the `uniqueItems` keyword. On the other hand, for the case of schemas using `uniqueItems` we can also show that these schemas are not definable in MSO. To illustrate this, recall Figure 3.1 that expresses documents representing complete binary trees. It is not difficult to show that this property cannot be accepted by a non-deterministic tree automata, and thus it cannot be expressed in MSO, as tree automata and MSO are equivalent in expressive power (Comon et al., 2007).

4. COMPLEXITY ANALYSIS

Now that we have delimited the expressive power of JSON Schema, we can use these results as the basis for a complexity analysis over the most common problems related to these documents. We begin by measuring the computational cost of checking if a document conforms to a schema, where we obtain results in both combined and data complexity. After this, we study the complexity of checking the existence of a JSON document that conforms to a given schema, providing tight complexity bounds and showing its inherent connection with tree automata theory.

4.1. The Validation Problem

In the basic API scenario, one could think of a verification layer composed by two schemas. The first schema could validate the incoming data and the second one could check the integrity of the output document. In this context, the most important question is to determine whether a JSON document J conforms to a schema S. We call this problem *JSON Schema validation* and it is formally defined as follows:

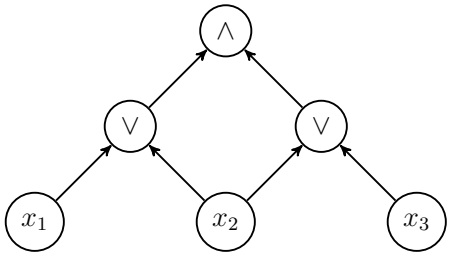
Problem:	JSCHVALIDATION(J, S)
Input:	JSON document J and schema S.
Question:	Does $J \models S$?

Since most JSON documents are used to transfer data between web applications, developing efficient algorithms for JSON Schema validation is of critical importance. It is thus important to understand the computational complexity of the schema validation problem, as this gives us a good starting point for the design of efficient validation algorithms. At this point, one could devise a simple procedure that works as follows: we process the document restriction by restriction, while checking conformance to the corresponding subschema in S. Based on this, we obtain our first result:

PROPOSITION 2. *The problem* JSCHVALIDATION(J, S) *is in* PTIME.

The details of the algorithm can be found at Appendix A.4. The running time is linear, because the correspondence to each keyword in JSON Schema can be checked in linear time (except for `uniqueItems`). If we keep `uniqueItems`, by simple inspection we obtain that our algorithm runs in $O(|J|^2 \cdot |S|)$.

Based on this result, we look towards establishing a tight bound for the validation problem. In fact, given the set of boolean operators that JSON Schema offers, one could perform a reduction from Monotone Circuit Value problem, which is known to be PTIME-complete (Goldschlager, 1977). Unfortunately, it is well-known that these problems are inherently sequential. In other words, the use of parallelism is not always going to be helpful to validate a document against a schema. The idea of the reduction is to show that for every circuit C and valuation τ one can construct a schema $S(C, \tau)$ and a JSON document $J(C, \tau)$ such that $\tau(C) = \texttt{true}$ if and only if, $J \models S$. We illustrate this procedure in Figure 4.1.




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

FIGURE 4.1. Schema for the circuit C with input values $\tau(x_1) = \tau(x_2) = \texttt{false}$ and $\tau(x_3) = \texttt{true}$.

In Figure 4.1 the `allOf` construct corresponds to the `AND` gate of the circuit, while the two `anyOf` subschemas simulate the `OR` gates. The input values are coded using `enum` in order to equal constants `true` and `false`. In addition, the document J just need to contain the value `true`. It is easy to devise that $J \vDash S \iff \tau(C) = \texttt{true}$, from which we obtain the following result:

PROPOSITION 3. *The problem* JSCHVALIDATION(J, S) *is* PTIME*-complete for the whole JSON Schema specification.*

The reduction and the detailed proof of Proposition 3 can be found at Appendix C.3. Now we turn our study to the context of data complexity. This analysis gives us better understanding of how the problem behaves in the context of web services, where most of the time we want to keep the schema fixed (recall the API example). The following result gives us a much better computing time for a practical use case.

PROPOSITION 4. *The problem* JSCHVALIDATION(J, S) *can be solved in* $O(|J|)$ *for any schema* S *not using the* `uniqueItems` *restriction.*

PROOF. Let J be a JSON document and S be a schema not using the `uniqueItems` restriction. We know that the validation problem can be described using an MSO formula φ_S and a tree structure $T(J)$. Now recall Courcelle's theorem (Courcelle, 1990), which states that every problem definable in Monadic Second Order logic can be solved in linear time on structures of bounded treewidth. Since the encoding of J has bounded tree width (as it is essentially a tree), Courcelle's theorem (Courcelle, 1990) applies, and we get the desired result. □

Note that the encoding of both the schema and the document can be computed in linear time.

4.2. The Satisfiability Problem

So far we have studied the most classic lineaments in the context of a semi structured data language. As we shown in Chapter 3, JSON Schema shares common features with both tree automata and logic. In general, a problem that is always present in these formalisms is the *satisfiability problem*. In its logic version the problem asks if it is possible to find a model for a propositional formula (George S. Boolos, 2007). In this section, we attempt to determine tight complexity bounds for this problem applied in the context of JSON Schema. Intuitively, given a schema S we want to check whether or not there exists a JSON document that conforms to it. If the answer is positive we say that S is *satisfiable*, in other case we say that it is *unsatisfiable*.

The importance of this problem becomes evident when developers try to design efficient algorithms for the generation of API documentation. For example, one could think of a program that given a schema, returns a fixed amount of JSON documents according to the schema. It is well known that these kinds of procedures always need to check the satisfiability of the input in order to provide the corresponding output.

Moreover, the problem is also necessary in topics of machine learning linked to JSON Schema. For instance, given a set of JSON documents, one could try to develop a procedure whose output corresponds to a schema that defines the initial set. Again, a satisfiability routine is needed to give a proper solution for this problem. Formally, we call this problem *JSON Schema satisfiability* and is defined as follows:

Problem:	JSCHSATISFIABILITY
Input:	A JSON Schema document S.
Question:	Is S satisfiable?

Some instances of this problem are shown in Example 9.

Example 9. Consider the following schemas:

$S_1 =$ {"type" : "integer", "minimum": 3, "maximum": 7}

$S_2 =$ {"type" : "integer", "minimum": 7, "maximum": 3}

In the first case S_1 is clearly satisfiable by any number between 3 and 7. On the contrary, S_2 cannot be satisfiable since there is no integer larger than 7 and smaller than 3.

The first question one could ask is how hard is this problem compared to its XML Schema version, which has been shown to be undecidable by Arenas, Fan, and Libkin (2002). However, since JSON Schema does not have some sort of key functionality, one would expect the problem to stay decidable. In fact, as we know that JSON Schema is captured by MSO, one can translate the satisfiability problem to its logic variant to get our first upper bound result.

PROPOSITION 5. *The problem* JSCHSATISFIABILITY *is decidable for schemas not using the* `uniqueItems` *restriction.*

PROOF. Let S be a JSON Schema document, from Theorem 1 we can build a formula φ_S in MSO such that S is satisfiable if and only if φ_S is satisfiable. Now we can adapt Büchi's theorem (Büchi, 1960) to construct a deterministic finite state automaton $\mathcal{A}_S$ for φ_S such that φ_S is satisfiable if and only if $L(\mathcal{A}_S) \neq \varnothing$. As we know that checking the emptiness for the language of a deterministic automaton can be computed in linear time, we get the desired result. □

The main problem of this result becomes evident when we take a closer look to the proof. In the demonstration we can devise that there could be a *non elementary* blow-up when we go from MSO to DFA. If we consider that φ_S is of the following form:

$$\varphi_S = (\exists \ldots \exists)(\forall \ldots \forall)(\exists \ldots \exists)\varphi$$

Büchi's theorem works over labeled trees, which is not the case for our MSO translation from JSON Schema. However, it is not hard to adapt Büchi's construction to our tree encoding.

our Büchi construction states that for each quantifier alternation we have to perform a determinization of our automaton $\mathcal{A}_S$, which implies an non elementary blow-up. Let $\mathcal{A}(\varphi)$ be the automaton for φ, the size of our automaton $\mathcal{A}_S$ will be bounded by

$$2^{2^{\cdot^{\cdot^{|\mathcal{A}(\varphi)|}}}}$$

In fact, it is known that converting MSO formulae into automata is inherently non elementary and the complexity cannot be lowered unless NP collapses to PTIME (Libkin, 2004).

The last result motivates a more exhaustive study for the satisfiability problem, always trying to chase a more tractable upper bound for it. In order to achieve this, we restrict the expressive power of JSON Schema by dividing it in different categories of schemas. We now define these expressiveness restrictions of the full specification. The first one encodes the sufficiency to express regular languages.

Definition 3 (Regular schema)**.** Given a JSON Schema document S, we say that S is *regular* if it makes use of either the `pattern` or `patternProperties` keyword.

However, definition 3 does not brace the capacity of JSON Schema to simulate tree automata. Intuitively, we can code these class of automata by the use of recursive calls; this fact motivates the following definition.

Definition 4 (Recursive schema)**.** Given a JSON Schema document S, we say that S is *recursive* if it makes use of the `definitions` section of the document.

Based on these definitions, we divide our analysis of the satisfiability problem by delimiting the study to three categories of schemas, each one of them with its own set of keywords. The categories, the sets of keywords and the reference to the proof of each result are summarized in table 4.1.

category	keywords allowed	lower bound	upper bound
recursive	**definitions**, `anyOf`, `allOf`, `not`, **pattern**, `required`, `properties`, **patternProperties**, `additionalProperties`	EXPTIME-hard Proposition 6	EXPTIME Theorem 3
non-recursive	`anyOf`, `allOf`, `not`, **pattern**, `required`, `properties`, **patternProperties**, `additionalProperties`	PSPACE-hard Proposition 7	PSPACE Proposition 8
non-regular non-recursive	`anyOf`, `allOf`, `not`, `enum`, `minLenght`, `maxLength`, `minimum`, `maximum`, `multipleOf`, `required`, `properties`, `additionalProperties`	NP-hard Proposition 9	NP Proposition 10

TABLE 4.1. The complexity of JSCHSATISFIABILITY for each subset of keywords.

As mentioned before, each set of keywords encodes common features such as the sufficiency to express regular expressions or to perform recursive calls in the schema. In the next chapters we prove these results taking in count that the keywords allowed are the ones expressed in our compacted grammar (Section 2.2).

4.2.1. The full case

As our starting point, we deal with the satisfiability problem in the context of the full specification of JSON Schema.

Since we can make use of the definitions section, it is possible combine self references to perform recursive calls to different schema definitions. Moreover, as we know that these recursive features are the basis to simulate tree automata we can get our first lower bound for the full case.

PROPOSITION 6. *The problem* JSCHSATISFIABILITY *is* EXPTIME*-hard for the class of recursive schemas, even if the schema just uses references, single enumeration and boolean combinations of schemas.*

PROOF. The reduction comes directly from Proposition 1. Consider $\mathcal{A}_1, \ldots, \mathcal{A}_n$ arbitrary tree automatons, we can construct $S(\mathcal{A}_1), \ldots, S(\mathcal{A}_n)$ schemas associated to $\mathcal{A}_1, \ldots, \mathcal{A}_n$ respectively. Now we can construct the following schema :

$$S = \{\texttt{"allOf":[} S(\mathcal{A}_1) \texttt{ ,..., } S(\mathcal{A}_n) \texttt{]}\}$$

Clearly S is satisfiable if and only if $L(\mathcal{A}_1 \cap \ldots \cap \mathcal{A}_n) \neq \varnothing$. Finally, as we know that intersection emptiness for non-deterministic tree automata is an EXPTIME-complete problem, we get the desired lower bound for the satisfiability problem. □

In terms of combined complexity, the result above gives us an approach of how hard is to determine the consistency of a given schema. Regardless of this, the reduction gives us a clue of how we could chase the upper bound for the problem.

From Proposition 5 we already know that the satisfiability must be decidable but assuming the cost of a non elementary blow-up. The issue lays in the inherently expensive cost of going from MSO to tree automata.

In order to chase a better upper bound, one could try to give a direct mapping from JSON Schema to some form of automata. In this context, the main problem is that the boolean operators can not be easily adapted to the classic definition of tree automata. Regardless of this, one could give a different approach using another model of automaton commonly used to deal with these complementations and intersections, *alternating tree automata* (Comon et al., 2007). Moreover, there is already a plenty of work related with automata running over ordered tree structures, which is not the case when we deal with JSON objects. Because of this, we remit our study for non array documents, by proposing a new model of tree automata over unordered trees, constructed specially to capture JSON Schema.

It is important to note that in this class of schemas, we do not consider the `multipleOf` and length restrictions, since they can be easy simulated by the use of a regular expression

Both string and numeric length restrictions.

(Sakarovitch, 2009). Similarly, the `enum` keyword can be simulated for both strings and objects. In the case of strings, we just need to force the string value by the use of a regular expression. For objects, we can always provide a schema that define a single object, just by combining the `required`, `properties` and `additionalProperties` keywords.

Now we provide the definition of our automata model for JSON Schema, recalling the tree structures defined in Chapter 3

$$\mathfrak{T} = \langle T_D, \Sigma^*, \prec_{ch}, \lambda, (R_L)_{L \subseteq \Sigma^*} \rangle$$

The next definition formalises an automaton model that runs over these structures.

Definition 5 (Quantified alternating tree automata)**.** A nondeterministic top-down **quantified alternating tree automata** (QATA) $\mathcal{A}$ over Σ is a tuple $(Q, \Sigma, \mathcal{I}, \Delta)$ where Q is a set of states, $\mathcal{I} \subseteq Q$ is the set of initial states and Δ is a finite set of transition rules of the following type:

$$q(R_\Sigma) \to \varphi$$

where $R_\Sigma \subseteq \Sigma^*$ is a regular language over Σ and $\varphi \in \mathcal{B}^+_{\exists\forall}(Q)$. Here $\mathcal{B}^+_{\exists\forall}(Q)$ is the set of propositional formulas over Q, where each state $q \in Q$ is preceded by a quantifier.

Example 10. Let $\Sigma = \{a, b\}$. Consider the automaton $\mathcal{A} = (Q, \Sigma, \mathcal{I}, \Delta)$ defined by $Q = \{q_0, q_1, q_2\}$, $\mathcal{I} = \{q_0\}$, and Δ=

$$q_0((ab)^*) \to \forall q_1 \wedge \exists q_2$$

$$q_1((aa)^*) \to true$$

$$q_2((aaa)^*) \to true$$

Note that we also use the $true$ and $false$ symbols also as valid formulas for the automaton. It is important to mention that the regular language in the left side of the rule is encoded as an alternating automaton. However, for ease of the exposition, the notation we use is the same as regular expressions.

Let v be a node labeled s with n children. A run of $\mathcal{A}$ over a tree structure $\mathfrak{T}$ is a mapping $\rho : T_D \to 2^Q$ such that if $\rho(v) = S$, then for each state q_S in S it must be true that:

$$\text{if } q_S(s) \to \psi \text{ then } \{\rho(u_1), ..., \rho(u_n)\} \vDash \psi \text{ where } s \prec_{ch} u_i \text{ for every } 1 \leq i \leq n.$$

Here the notion of satisfaction is defined inductively as follows. Let $\psi \in \mathcal{B}^+(Q)$ and $S \subseteq 2^Q$ such that $|S| = n$ we say that $S \vDash \psi$ if and only if:

- $\psi = true$
- if $\psi = \forall q$ then it must be true that $q \in S_i$ with $S_i \in S$ for every $1 \leq i \leq n$.
- if $\psi = \exists q$ then it must be true that $q \in S_i$ with $S_i \in S$ for some $1 \leq i \leq n$.
- if $\psi = \varphi_1 \wedge \varphi_2$ then it must be true that $S \vDash \varphi_1$ and $S \vDash \varphi_2$, for $\varphi_1, \varphi_2 \in \mathcal{B}^+(Q)$.
- if $\psi = \varphi_1 \vee \varphi_2$ then it must be true that $S \vDash \varphi_1$ or $S \vDash \varphi_2$, for $\varphi_1, \varphi_2 \in \mathcal{B}^+(Q)$.

A run ρ is successful if $q \in \rho(\epsilon)$ for some initial state $q \in \mathcal{I}$. Note that our definition corresponds to a nondeterministic model, where one string can match several rules from the left side of the relation.

For example, a run of $\mathcal{A}$ over the tree $T = abab(aa, aaaa, aaaaaa)$ is shown in Figure 4.2.

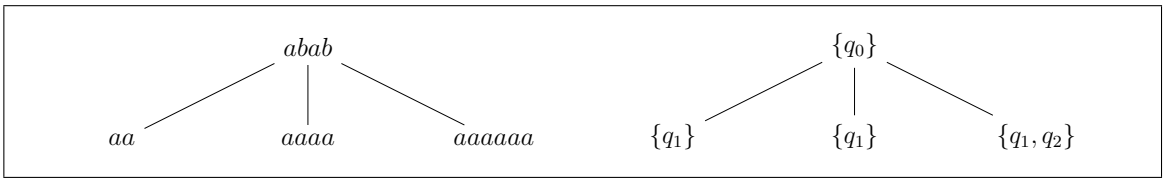


FIGURE 4.2. A run of a quantified alternating tree automaton

As promised, now we provide the translation between JSON Schema and quantified alternating tree automata. Recall our tree representation of JSON documents used in the translation to MSO. An example of this encoding is shown in Figure 4.3.

The main idea is to construct an quantified alternating tree automaton based on a JSON Schema document. To illustrate how this construction works consider the following schema S:

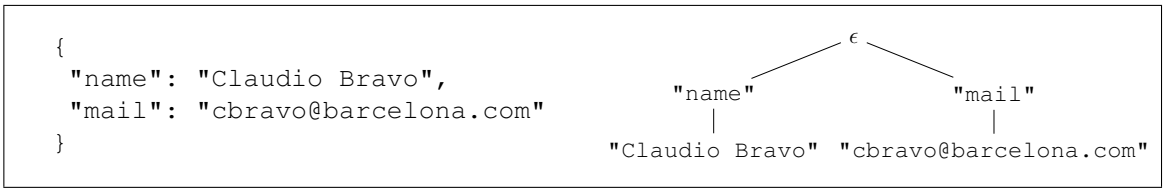


FIGURE 4.3. A JSON document and its coding as a tree.

```
{
 "type": "object",
 "required": ["name"]
 "properties": {
       "name": {"type": "string"},
       "mail": {"type": "string",
                 "pattern": "[A-z]*@barca.com"},
                      }
 }
```

The schema above is accepting soccer players from Barcelona FC, and the only required property is the name that must be a string. Moreover, if the document contains the keyword `"mail"`, then the value must comply to the regular expression described above. In particular, the document from Figure 4.3 validates against this schema, now we give a translation of S to QATA. Let $\mathcal{A}_S = (Q, \Sigma, \mathcal{I}, \Delta)$ defined by :

$$
\begin{array}{lll}
\mathcal{I} := \{q_0\} & Q := \{q_0, q_{name}, q_{mail}, q_{\Sigma^*}, q_{[A-z]*@barca.com}, q_\top, q_\bot\} & \\
\Delta := \{ & q_0(\epsilon) \to \exists q_{name} \wedge \forall q_{mail} & q_{\Sigma^*}(\Sigma^*) \to true \\
 & q_{name}(name) \to \exists q_{\Sigma^*} & q_\top(\Sigma^*) \to \forall q_\top \\
 & q_{name}(name^c) \to \exists q_\bot & q_\top(\Sigma^*) \to true \\
 & q_{mail}(mail) \to \exists q_{[A-z]*@barca.com} & q_\bot(\Sigma^*) \to \exists q_\bot \\
 & q_{mail}(mail^c) \to \forall q_\top & q_\bot(\Sigma^*) \to false) \\
 & q_{[A-z]*@barca.com}([A-z] * @barca.com) \to true & \}
\end{array}
$$

In the construction we use auxiliary states $q_\top, q_\bot$ to spread the accepting and rejecting states to the leafs of the run. Additionally, we use the universal quantifiers to handle the `properties` keyword where it is used as an implication (i.e. the mail property). Also, we use existential states to force the appearance of a keyword such as the case of the

"required" keyword. An example of an accepting run over the tree from Figure 4.3 of the automaton is shown in Figure 4.4.

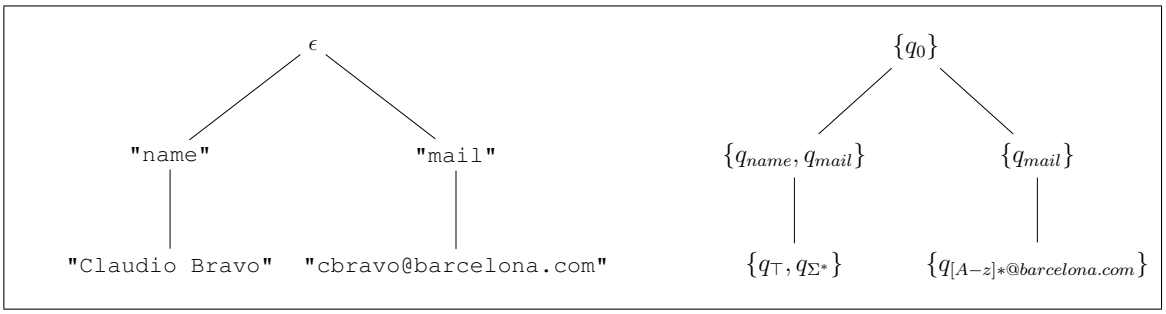


FIGURE 4.4. An accepting run of $\mathcal{A}_S$ over the document from Figure 4.3.

Although the procedure described above gives a mapping from one particular automaton , we provide the complete translation in Appendix C.5. We therefore obtain the following:

THEOREM 2. *For any non-array schema S there exists an QATA $\mathcal{A}_S$ such that for every JSON document J we have that $J \models S$ if and only if $T(J) \in L(\mathcal{A}_S)$.*

It is important to mention that the mapping can be performed in polynomial time, giving us a much better transition than the one from MSO to automata. Based on Theorem 2 we can easily deduce that a schema is satisfiable if and only if the language of its associated automaton is nonempty. This gives us the final step we need in order to get our EXPTIME upper bound. In fact, as we don't need to pass through MSO, we can avoid the non elementary blow-up. However, since we already proved hardness, EXPTIME is the best we dan do. Regardless of this, we obtain our desired result:

THEOREM 3. *The problem* JSCHSATISFIABILITY *is in* EXPTIME *for the class of recursive schemas.*

PROOF. The complete proof of Theorem 3 can be found on Appendix C.6. □

4.2.2. The non-recursive case

We continue our analysis by restricting the keywords allowed to be used in the schemas we are dealing with. In this case, our schemas cannot perform reference calls to the definitions section of the document. This implies that the height of our trees is alway bounded by the height of our schema. However, the horizontal language is still dependant on the `pattern` restrictions present in the schema. In fact, we can use the boolean operators the specification provides together with either the either the `pattern` or `patternProperties` restriction to simulate intersection emptiness of finite state automata, a problem known to be PSPACE-complete (Kozen, 1977). Based on this, we obtain our lower bound for the satisfiability problem for non-recursive schemas:

PROPOSITION 7. *The problem* JSCHSATISFIABILITY *is* PSPACE*-hard for the class of non-recursive schemas, even if the schema contains just the restrictions* `allOf`*,* `pattern`*,* `patternProperties`*,* `minProperties` *and* `additionalProperties`*.*

PROOF. Consider automatons $\mathcal{A}_1, \ldots, \mathcal{A}_n$; it is well known that every finite state automaton can be encoded into a regular expression, conserving the language of the automaton. Let $R(\mathcal{A}_1), \ldots, R(\mathcal{A}_n)$ be the regular expression associated with $\mathcal{A}_1, \ldots, \mathcal{A}_n$ respectively. Now we can construct the following schema :

$S =$ {"allOf":[
{"type": "string", "pattern":$R(\mathcal{A}_1)$}
,...,
{"type": "string", "pattern":$R(\mathcal{A}_n)$}
]}

Clearly S is satisfiable if and only if $L(\mathcal{A}_1 \cap \ldots \cap \mathcal{A}_n) \neq \varnothing$. Additionally, one can construct S with the use of the `patternProperties` constraint and obtain the same result:

$S =$ {"allOf":[
{"type": "object",
"additionaProperties": false,

```
 "minProperties": 1,
 "patternProperties":{"R(A_1)":{}}}
 ,...,
{"type": "object",
 "additionaProperties": false,
 "minProperties": 1,
 "patternProperties":{"R(A_n)":{}}}
                    ]}
```

Finally, as we know that finite automata intersection emptiness is known to be a PSPACE-complete problem, we conclude that JSCHSATISFIABILITY is PSPACE-hard. □

Let us continue with the corresponding upper bound. Since JSON schema definitions are not recursive, they are satisfied by JSON documents with a bounded number of nested objects, namely the maximum nesting of objects in the definition of the JSON schema.

Thus, for each non-recursive JSON Schema S, if n is the maximum nesting of objects in this Schema, then we know that all JSON documents conforming to S have at most n nested objects. Using this property, we can derive a coding scheme similar to that of (Benedikt, Fan, & Geerts, 2008) to transform JSON trees into strings, in a level-by-level fashion, taking into account the current depth of nesting within these objects.

As shown in (Benedikt et al., 2008) , the language of strings that represent a valid encoding of a JSON tree of depth at most n is a regular language. Thus, we can construct an automaton A_S that accepts all strings that represent encodings for JSON documents, and modify our QATA to work only over these types of strings. Note that because we allow boolean combinations we must keep the alternation in QATAs. However, it is well known that the emptiness problem for alternating string automata drops to PSPACE, and the same happens for the emptiness problem for QATAs restricted over strings. Thus, we obtain the upper bound for the class of non-recursive schemas:

PROPOSITION 8. *The problem* JSCHSATISFIABILITY *is in* PSPACE *for the class of non-recursive schemas.*

4.2.3. The non-regular non-recursive case

We conclude our analysis with the family of non-regular non-recursive schemas. These schemas are really important, since some of the most demanded web services communicate through these kind of documents. For example, Twitter API responses are formed according to the schema in Figure 4.5

```
{
    "type": "array",
    "items": {
       "type": "object",
       "required": ["favorited", "retweet_count", "text"],
       "properties": {
           "favorited": {"type": "boolean"},
           "retweet_count": {"type": "integer", "minimum": 0},
           "text": {"type": "string", "maxLength": 140}
       }
    }
 }
```

FIGURE 4.5. Twitter API response example.

In the example, the API gives as an output a list of the tweets posted by a Twitter account. The schema also defines information about the tweet that is contained in the response, such as the number of retweets and the "favorite" status. Note that this schema belongs to the non-regular non-recursive family, since it does not have neither pattern restrictions nor recursive calls.

For the analysis, as we don't have the `pattern` restriction, we decided to relax our grammar and include numeric types and length restrictions for strings that were not considered in the previous section.

Similarly as for the validation case, we start by providing the upper bound for the problem. Intuitively, as these schemas do not have any sort of recursion, the positives

Recall that numeric restrictions and length restrictions for strings can be simulated by NFA.

answers for the question should have a bounded size. As we know that the validation problem can be computed in PTIME, we can therefore obtain the NP upper bound for the problem, by providing the JSON document as a polynomial witness. However, if we have boolean combinations of schemas, it is not clear that these JSON documents are bounded by a polynomial. The Lemma below helps us to state this result:

LEMMA 1. *For each satisfiable schema S, there is a polynomial size JSON document J such that $J \models S$.*

The proof of Lemma 1 can be found at Appendix C.4. This result enable us to establish our desired upper bound for the satisfiability problem on the family of non-regular non-recursive schemas.

PROPOSITION 9. *The problem* JSCHSATISFIABILITY *is in* NP *for the class of non-regular non-recursive schemas.*

PROOF. Let S be a satisfiable JSON Schema document. From Lemma 1 we know that we could take a JSON document J as a polynomial witness for the problem. Finally, as we know that we can check the validation of J against S in polynomial time, we get that the satisfiability problem can be computed in NP. □

The above result gives us a much more tractable upper bound for the satisfiability problem. Besides, we confirm the intuition that the exclusion of regular expressions restrictions simplifies the problem in terms of computational complexity. However, the NP upperbound is not tractable and unfortunately, it is not avoidable. In fact, we could use JSON Schema boolean operators and perform a reduction from the classical SAT problem for propositional logic, which is known to be NP-complete (Cook, 1971).

PROPOSITION 10. *The problem* JSCHSATISFIABILITY *is* NP*-hard for the class of non-regular non-recursive schemas, even if the schema contains just the restrictions* `anyOf`*,* `allOF`*,* `not` *and* `required`*.*

PROOF. We now prove NP-hardness by reduction from SAT(φ). Let $P = \{p_1, \ldots, p_n\}$ and φ be a propositional formula over P. The central idea of the proof, is to construct a schema $S(\varphi)$ such that φ is satisfiable if and only if $S(\varphi)$ is satisfiable. The way this is done is by combining the boolean operators of JSON Schema with the `required` restriction for object schemas. Let us proceed inductively over the syntax of propositional logic:

- If $\varphi = p_i$, then $S(\varphi) = \{$`"type":"object","required"` $: [p_i]\}$ for $i \leq n$
- If $\varphi = \psi_1 \wedge \psi_2$, then $S(\varphi) = \{$`"allOf"` $: [S(\psi_1), S(\psi_2)]\}$
- If $\varphi = \psi_1 \vee \psi_2$, then $S(\varphi) = \{$`"anyOf"` $: [S(\psi_1), S(\psi_2)]\}$
- If $\varphi = \neg\psi$, then $S(\varphi) = \{$`"not"` $: S(\psi)\}$

To illustrate how the reduction works, consider $\varphi = (p \vee q) \wedge \neg r$, now we can construct $S(\varphi)$ as follows

```
{ "allOf": [
    {
      "anyOf":[
        {"type": "object", "required":["p"]},
        {"type": "object", "required":["q"]}
          ]
    },
    {
      "not": {"type": "object", "required":["r"]}
    }
  ] }
```

Note that the valuation for the formulas are coded as JSON objects such that the object has the keyword p_i if p_i is assigned to true in the valuation, for $i \leq n$. Also, it is important to mention that the reduction can be computed in linear time, since we have to iterate only once over the formula.

By simple construction, it is easy to devise that φ is satisfiable $\Leftrightarrow$ $S(\varphi)$ is satisfiable. □

5. CONCLUDING REMARKS

We provided a complete definition and formal analysis for the actual JSON Schema specification. The results we presented follow the guidelines established by Reutter et al. (2015) and give a much deeper analysis in both practical and theoretical aspects.

In terms of the specification, this work contributed by proposing a formal grammar for JSON Schema. This gives an unambiguous form for establishing the structure of such documents, a problem that was previously open. Moreover, we established formal semantics for the whole specification, defining how each restriction is validated in rigorous terms. It is important to mention that the work on the specification was done in collaboration with the IETF (Internet Engineering Task Force (IETF), 2014) and the online community of JSON Schema. Given the specification, we also conducted a formal analysis of both the expressiveness and computational complexity for JSON Schema.

In the analysis of expressiveness, we provided some results regarding to comparisons with classic formalisms. First, we proved that using a very restricted subset of the specification, we can simulate *tree automata*. Moreover, looking to bound the expressive power of JSON Schema, we proved that the complete specification can be captured by a very common formalism used in these structured data languages: *monadic second order logic*. This last result tells us that JSON Schema is not of equivalent expressivity as with XML Schema.

In terms of computational complexity, we focused our analysis on two problems: *validation* and *satisfiability*. In the validation case, we provided tight complexity bounds and proved that the problem remains PTIME-complete in terms of combined complexity. Moreover, if we take out array types, our translation to MSO gives us that the problem can be solved in linear time in terms of data complexity. Moreover, the mapping to MSO also give us some insight on the complexity for the satisfiability problem, showing that the problem stays decidable, in contrast to its XML variant (Arenas et al., 2002). However, we

also showed that this decidability is only reached by obtaining a non elementary blow-up in the size of the schema. Motivated by this, we provided a more complex analysis for the satisfiability problem of JSON Schema, looking to fill this gap with a tractable tight bound. The analysis was done by gradually decreasing the expressive power of JSON Schema.

We started by proving that in the most general case the problem is EXPTIME-complete. This last result gives us a much better upper bound than the one obtained directly from the translation to logic, and it is done by introducing a new computational model: *quantified alternating tree automata*. Then, if we restrict the expressiveness of our schemas by taking out the recursive operators, the problem proves to be PSPACE-complete. Finally, if we also take out the regular expressions, the problem becomes much easier and turns out to be NP-complete.

In practical terms, our complexity results give us some important consequences in the adoption of JSON Schema as a standard.

On one side, the linear upper bound for the validation problem shows that it is feasible to implement JSON Schema in the top layer of an API. This point is crucial, given the popularity this protocol has gained in the last few years. However, the fact that the problem is PTIME-complete tells us in a way that it is not always helpful to us parallelism for the validation of a single document.

On the other side, our satisfiability results have potential application in the context of documentation automation, where SAT algorithms have been shown to be the base for these purposes. The importance of this problem becomes evident when examining the documentation of the most complex APIs available on the web. Most of these APIs describe their protocol based on a list of examples for the user. The main idea is that this list could be generated automatically by an algorithm that runs over the input (output) schemas of the web service. In this context, our EXPTIME algorithm could serve as the basis for this purpose. Even if the upper bound is exponential, we also showed that it

could be reduced to NP for more restricted classes of schemas, which are in fact, the most common cases we find in the web.

In future work, we would like to point out the potential use of JSON Schema in the context of document oriented databases such as (MongoDB, 2015; CouchDB, 2015; RethingDB, 2015) among others. Most of these DBMS work over JSON documents as storage units. An interesting problem here would be the implementation of integrity constraints in the specification of JSON Schema, to serve as the schema basis for these non-relational database models.

References

Arenas, M., Fan, W., & Libkin, L. (2002). What's hard about xml schema constraints? In *13th international conference on database and expert systems applications.*

Benedikt, M., Fan, W., & Geerts, F. (2008). XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, *55*(2), 8.

Berman, J. (2015). *JSON Schema Test Suite.* `https://github.com/json-schema/JSON-Schema-Test-Suite`.

Berners-Lee, T. (2005, January). *Uniform resource identifier (uri): Generic syntax.* https://tools.ietf.org/html/rfc3986.

Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format.

Büchi, J. R. (1960). *Weak second-order arithmetic and finite automata* (Tech. Rep.). The University of Michigan.

Comon, H., Dauchet, M., Gilleron, R., Loding, C., Jacquemard, F., Lugiez, D., ... Tommasi, M. (2007). *Tree automata techniques and applications*.

Cook, S. A. (1971). *The complexity of theorem-proving procedures.*

Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2001). *Introduction to algorithms* (2nd ed.). McGraw-Hill Higher Education.

Courcelle, B. (1990). The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*.

Galiegue, F., & Zyp, K. (2013). *Json schema: Core definitions and terminology.* `http://json-schema.org/latest/json-schema-core.html`.

George S. Boolos, R. C. J., John P. Burgess. (2007). *Computability and logic*. Cambridge.

GitHub, I. (2013a). *Heroics: Ruby HTTP client for APIs represented with JSON schema.* https://github.com/interagent/heroics.

GitHub, I. (2013b). *Prmd: JSON Schema tools and documentation generation for HTTP APIs.* https://github.com/interagent/prmd.

GitHub, I. (2013c). *Schematics: A Go point of view on JSON Schema.* https://github.com/interagent/schematic.

Goldschlager, L. M. (1977, July). The Monotone and Planar Circuit Value Problems Are Log Space Complete for P. *SIGACT News*, *9*(2), 25–29. Retrieved from http://doi.acm.org/10.1145/1008354.1008356 doi: 10.1145/1008354.1008356

Google. (2015). *Google API Discovery Service.* https://developers.google.com/discovery/.

Hilaiel, L. (2015). *Orderly.* http://orderly-json.org/.

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

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

json-schema.org: The home of json schema. (2016). http://json-schema.org/. Retrieved from http://json-schema.org/

Kozen, D. (1977). Lower bounds for natural proof systems. *SFCS '77 Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 254-266.

Libkin, L. (2004). *Elements of finite model theory*. Springer.

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

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

MongoDB Inc. (2015). *The MongoDB3.0 Manual.* `https://docs.mongodb.org/manual/`.

Neven, F. (2002). Automata theory for xml researchers. *SIGMOD Record Database Principles Column*.

Papadimitriou, C. H. (1994). *Computational complexity*. Addison-Wesley.

Rao, L. (n.d.). *Twitter seeing 6 billion api calls per day, 70k per second.*

RethinkDB: The open-source database for the realtime web. (2015). `https://www.rethinkdb.com/`.

Reutter, J. L., Ugarte, M., & Vrgoc, D. (2015, May). Satisfiability of json schema and instance validation. Alberto Mendelzon International Workshop on Foundations of Data Management(AMW).

Sakarovitch, J. (2009). *Elements of automata theory*. Cambridge.

Sporny, M., Kellogg, G., & Lanthaler, M. (2014, January). *JSON-LD 1.0: A JSON-based Serialization for Linked Data.* `http://www.w3.org/TR/json-ld/`.

Swagger: The World's Most Popular Framework for APIs. (2015). `http://swagger.io/`.

The Apache Software Foundation. (2015). *Apache CouchDB*. http://couchdb.apache.org/.

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

The RAML Workgroup. (2015). *RAML: RESTful API Modeling Language*. http://raml.org/.

APPENDIX A. ADDITIONAL TESTS AND CODE

A.1. Four documents and four schemas used for testing

```
{
"tests":[
{
"schema":{"uniqueItems":true},
"document":[{"a":3,"b":4},{"b":4,"a":3}]
},
{
"schema":{"definitions":{"a":{"type":"string"}},
"$ref":"#/definitions/a","type":"integer"},
"document":"hola"
},
{
"schema":{"required":["a","b"],"multipleOf":3},
"document":4},
{
"schema":{"type":"object","properties":{"a":{"type":"string"}},
"required":["a"],"dependencies":{"a":{"additionalProperties":false}}},
"document":{"a":"asdf"}
},
{
"schema":{"definitions":{"a":{"$ref":"#/definitions/a"}},
"anyOf":[{"$ref":"#/definitions/a"},{"type":"string"}]},
"document":"hola"}]
}
```

A.2. Script used to test border cases

A.2.1. Tester

```
from jsonschema import validate, exceptions
from subprocess import check_output
from jsonspec.validators import load
from jsonspec.validators.exceptions import ValidationError
import requests
import urllib
import json

_JAVA_URL = 'http://json-schema-validator.herokuapp.com/process/index'
_MARK_DICT = {1: '\\CheckmarkBold',
              0: '\\XSolidBrush',
              -1: '\\Cross'}


def python_validate(schema, document):
    try:
        validate(document, schema)
        return True
    except exceptions.ValidationError:
        return False


def is_my_json_valid(schema, document):
    command = ["node",
               "is-my-json-valid.js",
               json.dumps(schema),
               json.dumps(document)]
    result = str(check_output(command))[2:-3]
    if result == 'true':
        return True
    elif result == 'false':
        return False


def java_json_schema_validator(schema, document):
    schema_input = urllib.parse.quote_plus(json.dumps(schema))
```

```
    document_input = urllib.parse.quote_plus(json.dumps(document))
    data = 'input=\%s&input2=\%s' \% (schema_input, document_input)
    result = requests.post(_JAVA_URL,
                           data=data)
    return json.loads(result.text)['valid']


def python_validate_2(schema, document):
    validator = load(schema)
    try:
        validator.validate(document)
        return True
    except ValidationError:
        return False


def ruby_json_schema(schema, document):
    command = ["ruby",
               "json-schema.rb",
               json.dumps(schema),
               json.dumps(document)]
    result = str(check_output(command))[2:-3]
    if result == 'true':
        return True
    elif result == 'false':
        return False


def ruby_json_schema_2(schema, document):
    command = ["ruby",
               "ruby_json_schema.rb",
               json.dumps(schema),
               json.dumps(document)]
    result = str(check_output(command))[2:-3]
    if result == 'true':
        return True
    elif result == 'false':
        return False
```

```
def validate_tests(validators, filepath):
    tests = json.loads(open(filepath).read())["tests"]
    counter = 1
    results = []

    for test in tests:
        schema = test["schema"]
        document = test["document"]
        print('\n--------Test \%d:--------' \% counter)
        print('#  Schema:')
        print_json(schema)
        print('#  Document:')
        print_json(document)

        test_results = {}
        for name, function in validators.items():
            try:
                test_results[name] =1 if function(schema, document) else 0
            except:
                test_results[name] = -1
            # print('\%s: \%r' \% (name, test_results[name]))

        results.append(test_results)
        counter += 1

    print('\n\n---------- RESULTS ----------')

    print('\nValidators:\n')

    counter = 0
    for validator in validators:
        print('V\%d: \%s' \% (counter, validator))
        counter += 1

    first_row = '\\hline & '
    for i in range(counter):
        first_row += 'V\%d & ' \% i

    print(first_row[:-3] + '\\\\')
    counter = 0
```

```
    for result in results:
        result_str = "\\hline test \%d: & " \% counter
        for name in result:
            result_str += ' \%s &' \% _MARK_DICT[result[name]]
        print(result_str[:-2] + '\\\\')
        counter += 1


def print_json(parsed_json):
    print(json.dumps(parsed_json, indent=4))

if __name__ == '__main__':
    validators = {
        "jsonschema (python)": python_validate,
        "is-my-json-valid (javascript)": is_my_json_valid,
        "json-schema-validator (java)": java_json_schema_validator,
        "json-schema-validator (python)": python_validate_2,
        "json-schema (ruby)": ruby_json_schema,
        # "ruby: json-schema_2": ruby_json_schema_2,

    }
    filepath = 'tests.json'
    validate_tests(validators, filepath)
```

A.2.2. is-my-json-valid.js

```
var validator = require('is-my-json-valid')
var validate = validator(JSON.parse(process.argv[2]))
console.log(validate(JSON.parse(process.argv[3])))
```

A.2.3. json-schema.rb

```
require "json-schema"
require '"json"
result = JSON::Validator.validate(JSON.parse(ARGV[0]), ARGV[1])
puts(result)
```

A.2.4. ruby_json_schema.rb

```
require "json"
require "json_schema"
schema_data = JSON.parse(ARGV[0])
```

```
schema = JsonSchema.parse!(schema_data)
data = JSON.parse(ARGV[1])
begin
schema.validate!(data)
puts('true')
rescue
puts('false')
end
```

A.3. Reduction from $\mathcal{A}$ to JSON Schema

```
{
  "definitions": {
    "q1": {
      "type": "object",
      "properties": {
        "and": {
          "anyOf": [{
                        "type": "object",
                          "properties": {
                             "lChild": {"$ref": "#/definitions/q1"},
                             "rChild": {"$ref": "#/definitions/q1"}},
                           "additionalProperties": false,
                           "required": ["lChild","rChild"]}
                            ] },
        "or": {
          "anyOf": [
                        {"type": "object",
                          "properties": {
                              "lChild": {"$ref": "#/definitions/q1" },
                              "rChild": {"$ref": "#/definitions/q1"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]},
                        {"type": "object",
                          "properties":{
                               "lChild": {"$ref": "#/definitions/q1" },
                               "rChild": {"$ref": "#/definitions/q0"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]},
                        {"type": "object",
                          "properties": {
                               "lChild": {"$ref": "#/definitions/q0" },
                               "rChild": {"$ref": "#/definitions/q1"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]}
                          ]},
          "1": {"enum": [null]}
        },
        "additionalProperties": false,
        "minProperties": 1
      },
      "q0": {
        "type": "object",
        "properties": {
          "or": {
            "anyOf": [{
                        "type": "object",
                          "properties": {
                             "lChild": {"$ref": "#/definitions/q0"},
                             "rChild": {"$ref": "#/definitions/q0"}},
                           "additionalProperties": false,
                           "required": ["lChild","rChild"]}
                            ] },
          "and": {
            "anyOf": [
                        {"type": "object",
                          "properties": {
                              "lChild": {"$ref": "#/definitions/q0" },
                              "rChild": {"$ref": "#/definitions/q0"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]},
                        {"type": "object",
                          "properties":{
                               "lChild": {"$ref": "#/definitions/q1" },
                               "rChild": {"$ref": "#/definitions/q0"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]},
                        {"type": "object",
                          "properties": {
                               "lChild": {"$ref": "#/definitions/q0" },
                               "rChild": {"$ref": "#/definitions/q1"}},
                          "additionalProperties": false,
                          "required": ["lChild","rChild"]}
                          ]},
            "0": {"enum": [null]}
          },
          "additionalProperties": false,
          "minProperties": 1
        }
      },
      "anyOf": [
        {
          "$ref": "#/definitions/q1"
        }
      ]
    }
```

FIGURE A.1. JSON Schema document for automaton $\mathcal{A}$.

A.4. JSON Schema validation algorithm

Algorithm 1 Validation algorithm for JSON Schema.

```
1:  procedure VALIDATE(S, J)
2:      if S["type"] == "string" then
3:          if VALIDATESTRING(S, J) then
4:              return TRUE
5:          else
6:              return FALSE
7:          end if
8:      end if
9:      if S["type"] == "object" then
10:         if VALIDATEOBJECT(S, J) then
11:             return TRUE
12:         else
13:             return FALSE
14:         end if
15:     end if
16:     if S["type"] == "array" then
17:         if VALIDATEARRAY(S, J) then
18:             return TRUE
19:         else
20:             return FALSE
21:         end if
22:     end if
23:     if S["allOf"] == [S_1, ..., S_n] then
24:         for S_i in [S_1, ..., S_n] do
25:             if !VALIDATE(S_i, J) then
26:                 return FALSE
27:             end if
28:         end for
29:         return TRUE
30:     end if
31:     if S["anyOf"] == [S_1, ..., S_n] then
32:         for S_i in [S_1, ..., S_n] do
33:             if VALIDATE(S_i, J) then
34:                 return TRUE
35:             end if
36:         end for
37:         return FALSE
38:     end if
39:     if S["not"] == S' then
40:         if !VALIDATE(S', J) then
41:             return TRUE
42:         else
43:             return FALSE
44:         end if
45:     end if
46:     if S["enum"] == [J_1, ..., J_n] then
47:         if J ∈ [J_1, ..., J_n] then
48:             return TRUE
49:         else
50:             return FALSE
51:         end if
52:     end if
```

Algorithm 2 Subroutine for validating string documents.

```
1:  procedure VALIDATESTRING(S, J)
2:      if J is not string then
3:          return FALSE
4:      end if
5:      if S["pattern"] == regex then
6:          if J ∉ L(regex) then
7:              return FALSE
8:          end if
9:      end if
10:     return TRUE
```

Algorithm 3 Subroutine for validating object documents.

```
1:  procedure VALIDATEOBJECT(S, J)
2:      if J is not object then
3:          return FALSE
4:      end if
5:      if S["required"] == [k_1, ..., k_n] then
6:          for key in [k_1, ..., k_n] do
7:              if J[key] == ∅ then
8:                  return FALSE
9:              end if
10:         end for
11:     end if
12:     for key in J do                                   ▷ Validate each keyword in J
13:         for string ∈ S["properties"] do
14:             if string == key then
15:                 if !VALIDATE(S["properties"][key], J[key]) then
16:                     return FALSE
17:                 end if
18:             end if
19:         end for
20:         for regex in S["patternProperties"] do
21:             if key ∈ L(regex) then
22:                 if !VALIDATE(S["patternProperties"][regex], J[key]) then
23:                     return FALSE
24:                 end if
25:             else
26:                 if key ∉ S["properties"] then
27:                     if !VALIDATE(S["additionalProperties"][key], J[key]) then
28:                         return FALSE
29:                     end if
30:                 end if
31:             end if
32:         end for
33:     end for
34:     return TRUE
```

Algorithm 4 Subroutine for validating array documents.

```
1:  procedure VALIDATEARRAY(S, J)
2:      if J is not array then
3:          return FALSE
4:      end if
5:      if S["minItems"] == n then
6:          if |J| < n then
7:              return FALSE
8:          end if
9:      end if
10:     if S["maxItems"] == n then
11:         if |J| > n then
12:             return FALSE
13:         end if
14:     end if
15:     if S["Items"] == S' then
16:         for J_i in J do
17:             if !VALIDATE(S', J_i) then
18:                 return FALSE
19:             end if
20:         end for
21:     end if
22:     if S["Items"] == [S_1, ..., S_n] then
23:         for i ≤ min(|J|, n) do
24:             if !VALIDATE(S_i, J_i) then
25:                 return FALSE
26:             end if
27:         end for
28:     end if
29:     if S["uniqueItems"] == true then
30:         for J_i in J do
31:             for J_k in J do
32:                 if i ≠ k & J_i == J_k then
33:                     return FALSE
34:                 end if
35:             end for
36:         end for
37:     end if
38:     return TRUE
```

APPENDIX B. FORMAL SPECIFICATION

B.1. Syntax

B.1.1. Notation

The Formal Grammar in this specification is given using a simple, visual-based Extended Backus-Naur Form notation(The International Organization for Standardization (ISO), 1996), that we define below. Each rule in the grammar defines one symbol, in the form:

symbol `:=` `expression`

For readability we always write non-terminal symbols in blackened font, such as **JSch**or **strRes**. The expression on the right hand side of these rules may match more than one string, and is constructed according to the following operators:

- **string**: any non-blackened string that does not use `)`, `(`, `|` or `?` matches precisely against the string.

We also use brackets, as in `(`**`expression`**`)`, to specify that the expression inside them is a unit. We can combine units using the following operators:

- **`E?`**: Optional **`E`**, matches **`E`** or nothing.
- **`A|B`**: **`A`** or **`B`**, matches **`A`** or **`B`**.
- **`AB`**: **`A`** concatenated with **`B`**, matches **`A`** followed by **`B`**. This operator has higher precedence over `|`.
- **`E`***: Matches zero or more occurrences of **`E`**. Also has a higher precedence over `|`.

B.1.2. Grammar

Formally we define a JSON Schema Document as a set of definitions and a schema. Each schema is treated as a set of restrictions that may apply to one or more types. To keep a clean and tidy grammar we divide each restriction in different sections, but as every grammar, the document is defined by the union of all these nested variables.

B.1.2.1. Top layer structure

Let **JSDoc** be an arbitrary JSON Schema Document. We can define its syntax using the following grammar:

JSDoc := { (**id,**)? (**defs,**)? **JSch** }
id := "id": **"uri"**
defs := "definitions": { **kSch**(, **kSch**)* }
kSch := texttt **string**: { **JSch** }
JSch := **strSch** | **numSch** | **intSch** | **boolSch** | **nullSch** | **objSch** | **arrSch** | **combSch** | **refSch**

Here each **string** is representing a keyword that must be unique in the nest level that is occurs. Next we specify the remaining schema types: **strSch**, **numSch**, **intSch**, **boolSch**, **nullSch**, **objSch**, **arrSch**, **combSch** and **refSch**.

B.1.2.2. String schemas

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

Here each **strRes** must be different from each other. Besides, **n** is a natural number and **RegExp** is a regular expression.

B.1.2.3. Numeric schemas

intSch := `"type": "integer"` (`,` **numRes**)*

numSch := `"type": "number"` (`,` **numRes**)*

numRes := **min** | **exMin** | **max** | **exMax** | **mult**

min := `"minimum":` **r**

exMin := `"exclusiveMinimum":` **bool**

max := `"maximum":` **r**

exMax := `"exclusiveMaximum":` **bool**

mult := `"multipleOf":` **r** ($\mathbf{r} \geq \mathbf{0}$)

Here each **numRes** must be different from each other. Besides, **r** is a decimal number and **bool** is either true or false.

B.1.2.4. Boolean schemas

boolSch := `"type": "boolean"`

B.1.2.5. Null schemas

nullSch := `"type": "null"`

B.1.2.6. Object schemas

objSch := `"type": "object"` (`,` **objRes**)*

objRes := **prop** | **addPr** | **req** | **minPr** | **maxPr** | **deps** | **patPr**

prop := `"properties": {` **kSch**(`,` **kSch**)* `}`

kSch := **string**`:` `{` **JSch** `}`

addPr := `"additionalProperties": (` **bool**| `{` **JSch** `})`

req := `"required": [` **string** (`,` **string**)* `]`

minPr := `"minProperties":` **n**

maxPr := `"maxProperties":` **n**

deps := `"dependencies": (` **depSch** | **depArr** `)`

depSch := `{` **kSch** (`,` **kSch**)* `}`

depArr := `{` **kArr** (`,` **kArr**)* `}`

kArr := **string**: [**string** (, **string**)*]

patPr := "patternProperties": { **patSch**(, **patSch**)* }

patSch := **RegExp**: { **JSch** }

Here each **objRes** must be different from each other. Besides, **n** is a natural number, **bool** is either true or false and **RegExp** is a regular expression. As above, each **string** is representing a keyword that must be unique in the nest level that is occurs.

B.1.2.7. Array schemas

arrSch := "type": "array" (, **arrRes**)*

arrRes := **items** | **addItems** | **minIt** | **maxIt** | **unique**

items := (**itemo** | **itema**)

itemo := "items": { **JSch** }

itema := "items": [{**JSch**} (, {**JSch**})*]

minIt := "minItems": **n**

maxIt := "maxItems": **n**

unique := "uniqueItems": **bool**

Here each **arrRes** must be different from each other. Besides, **n** is a natural number and **bool** is either true or false.

B.1.2.8. Boolean combination schemas

combSch := **not** | **allOf** | **anyOf**| **oneOf** | **enum**

not := "not": { **JSch** }

allOf := "allOf": [{**JSch**} (, {**JSch**})*]

anyOf := "anyOf": [{**JSch**} (, {**JSch**})*]

oneOf := "oneOf": [{**JSch**} (, {**JSch**})*]

enum := "enum": [**Jval** (, **Jval**)*]

Here **Jval** is either a string, number, array, object, bool or a null value. Moreover each **Jval** must be different from each other(otherwise they would be superfluous).

B.1.2.9. Reference schemas

refSch := `"$ref": "uriRef"`

uriRef := `( address )? ( #/ JPointer )?`

JPointer := `( / path )`

path := `( escaped | unescaped )`

escaped := `~0 | ~1`

Where **unescaped** can be any character except for **/** and ~. Also, **address** corresponds to any URI that does not use the **#** symbol, or more precisely to any URI-reference constructed using the following grammar, as defined in the official standard(Berners-Lee, 2005):

address := `( scheme )? hier-part ( ? query )`

B.2. Semantics

In this section we present a formal specification of how JSON Schema restrictions are validated against an arbitrary JSON Document. But before specifying the semantics for these validation instances we must define a couple of structures first.

B.2.1. JSON Reference

If **path** is a JSON pointer, we say that **rep(path)** is the string resulting of replacing first each character $\sim$`1` by `/` and then each character $\sim$`0` by $\sim$.

Given a JSON document J that is an object, we use $J[k]$ (for a string k) to represent the value of the key value pair in J whose key is k . Likewise, if J is an array, then $J[n]$ (for a natural number n) corresponds to the n-th element of J .

Let J be a JSON document. JSON Pointers are intended to extract a part of J that is specifically indexed by the pointer. Formally, we define the function **EVAL** that takes a JSON Document J and a JSON Pointer $JPointer$ and delivers a subset of J:

EVAL$(J, JPointer)$ returns:

- J, if $JPointer$ is the empty string, or
- **EVAL**$(J[$**rep**$(key)], JP)$, if J is an object, **rep**(key) appears in J and $JPointer$ is of the form JP/key, or
- **EVAL**$(J[n], JP)$, if $JPointer$ is an array with at least $n + 1$ objects and $JPointer$ is of the form JP/n, where n is the base-10 representation of a natural number, or
- an error in any other case (for example when the pointer asks for a key that is not in J).

Let R be a JSON Reference of the form `"$ref":` $uriRef$. We extend the function **EVAL** to work with arbitrary JSON references. We do it as follows.

EVAL(J, R) returns:

- **EVAL**$(J, JPointer)$, if $uriRef$ is of the form `#/`$JPointer$, or
- S, if $uriRef$ does not contains the symbol # and the address in $uriRefsuccesfully$ retrieves the schema S, or

- **EVAL**$(S, JPointer)$, if $uriRef$ is of the form address #/$JPointer$ and the address in $uriRefsuccesfully$ retrieves the schema S, or
- an error in any other case (for example when the address retrieves something that is not a schema).

Let R be again a JSON Reference and J a JSON document. Assume that the JSON schema document that contains R is S. Then we say that J validates against R under S if **EVAL**(S, R) returns a schema (not an error) and J validates against **EVAL**(S, R).

B.2.2. Validation

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

B.2.2.1. References and boolean combination of schemas

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

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

B.2.2.2. Strings schemas

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

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

- p is `"pattern": e` and J is a string that belongs to the language of the expression `e`.

B.2.2.3. Numeric schemas

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

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

B.2.2.4. Boolean schemas

Let S be a boolean schema. Then $J \models S$ if J is either `true` or `false`.

B.2.2.5. Null schemas

Let S be a null schema. Then $J \models S$ if J is the `null` value.

B.2.2.6. Objects schemas

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

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

  Remark. If the keyword matches more than one pattern property then it has to satisfy all the schemas involved.

- p is `"required":` $[k_1, \ldots, k_m]$ and for each $1 \leq j \leq m$ we have that J has a pair of the form $k_j : v$.
- p is `"additionalProperties":` `false` and for each pair $k : v$ in J, either S contains `"properties":` $\{k_1 : S_1, \ldots, k_m : S_m\}$ and $k = k_j$ for some $1 \leq j \leq m$, or S contains `"patternProperties":` $\{$`"`e_1`"`$: S_1, \ldots,$ `"`e_m`"`$: S_m\}$ and k belongs to the language of e_j, for some $1 \leq j \leq m$.
- p is `"additionalProperties":` S' and for each key-value pair $k' : j'$ in J, with k' not in S[properties] and k' not matching any of the expressions in S[patternProperties], we have that j' validates against S' `"properties":` $\{k_1 : S_1, \ldots, k_m : S_m\}$ and $k = k_j$ for some $1 \leq j \leq m$, or S contains `"patternProperties":` $\{$`"`e_1`"`$: S_1, \ldots,$ `"`e_m`"`$: S_m\}$ and k belongs to the language of e_j, for some $1 \leq j \leq m$.
- p is `"minProperties":`n and J has at least n key-value pairs.
- p is `"maxProperties":`n and J has at most n key-value pairs.
- p is of the form `"dependencies":` $\{k_1 : [l_{1,1}, \ldots, l_{1,m_1}], \ldots, k_n : [l_{n,1}, \ldots, l_{n,m_n}]\}$ and if k_i appears in J then every keyword in $[l_{i,1}, \ldots, l_{i,m_i}]$ appears in J.
- p is of the form `"dependencies":` $\{k_1 : S_1, \ldots, k_m : S_m\}$ and if k_i appears in J then it must be true that $J \models S_i$.

B.2.2.7. Arrays schemas

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

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

APPENDIX C. PROOFS

C.1. Proof of Proposition 1

PROOF. We proceed to construct procedure to encode each NFTA into a JSON Schema document. Let $\mathcal{A} = (Q, \mathcal{F}, \mathcal{I}, \Delta)$ be a top-down NFTA over binary trees where Q is a set of states, $I \subseteq Q$ is a set of initial states and $\Delta \subseteq Q \times \Sigma \times Q \times Q$ is the transition relation(i.e. set of rules). Consider the following encoding of $\mathcal{A}$ into $S(\mathcal{A})$:

(1) For each state $q \in Q$ append to S a new schema definition of the form:

```
"q": {
  "type": "object",
  "additionalProperties": false,
  "minProperties": 1,
  "properties": {P_q}
  }
```

Here P_q represents the set of symbols for outgoing transitions of q.

(2) For each symbol $\sigma \in \mathcal{F}$ if there is a rule of the form $q \to \sigma$ in Δ, then add the following property to P_q:

```
"σ": { "enum": [null] }
```

(3) For each symbol $\sigma \in \mathcal{F}$ if there is a rule of the form $q(\sigma) \to (q_1, q_2)$ in Δ, then add the following property to P_q:

```
"σ": { "anyOf": [A_q(σ)] }
```

Here $A_{q(\sigma)}$ is an array representing the possible states reachable from q reading σ.

(4) For each pair $(q_1, q_2) \in Q \times Q$ if there is a rule $q(\sigma) \to (q_1, q_2)$ in Δ, then add the following schema to $A_{q(\sigma)}$:

```
{
 "type": "object",
 "additionalProperties": false,
 "required": ["lChild", "rChild"],
 "properties": {
                "lChild": "ref": "#/definitions/q1"
                "rChild": "ref": "#/definitions/q2"
                 }
 }
```

(5) Finally add the following schema to the body of the schema

```
"anyOf": [
{"ref": "#/definitions/q_1"},
.
.
.
{"ref": "#/definitions/q_n"}
]
```

Where $q_1, ..., q_n$ are the initial states of $\mathcal{A}$.

Let T be a binary tree and $\mathcal{A}$ a NFTA, by simple construction we obtain that $T \in L(\mathcal{A})$ if and only if, $J(T) \models S(\mathcal{A})$. □

C.2. Proof of Theorem 1

PROOF. Let J be a JSON document and JS a JSON Schema document, now we prove that there exists a formula in monadic second order logic φ_{JS} such that

$$J \models JS \text{ if and only if } T(J) \models \varphi_{JS}$$

where $T(J)$ is the tree representation of J described in section 3.3.

First of all, let us define the structures and the interpretation we use to model the theory of JSON documents. Let $\mathcal{L}$ be the following vocabulary:

$$\mathcal{L} = \langle \prec_{ch}, P_{root}, \lambda, (R_L)_{L \subseteq \Sigma^*} \rangle$$

As we know that we are working with trees we can consider the following interpretation of $\mathcal{L}$:

- $\prec_{ch} \subseteq \mathbb{N}^* \times \mathbb{N}^*$ is the child relation over the tree domain
$$v \prec_{ch} v' \text{ if and only if } s' = s \cdot i \text{ for some } i \in \mathbb{N}.$$
- $P_{root} \subseteq \mathbb{N}^*$ is the root predicate, let v be a node
$$P_{root}(v) \text{ if and only if } v \text{ is the root of the tree.}$$
- $\lambda : \mathbb{N}^* \to \Sigma^*$ is the labeling of the tree, we use it to assing keywords and values to the document. Let v be a node and s be a string
$$\lambda(v) = s \text{ if and only if the label } s \text{ is assigned to the node } v.$$
- $(R_L)_{L \subseteq \Sigma^*}$ is a predicate to represent the pertinence of a string in a regular language L. Let s be a string, we define this formally as
$$R_L(s) \text{ is true if and only if } s \text{ is in the language } L.$$

Finally, we represent the $\mathcal{L}$-structures that model each JSON document as :

$$\mathfrak{T} = \langle T_D \cup \Sigma^*, \prec_{ch}, P_{root}, \lambda, (R_L)_{L \subseteq \Sigma^*} \rangle$$

As mentioned, we first need to define the theory of JSON documents. Formally, this is done by constructing a formula φ_{JSON} such that every JSON document satisfies it. We construct this by parts, let us start with the root, consider the following formula to define a parentless node:

We do this for the reason that we are working over tree structures.

$$\varphi_{no\ child} = \forall y(y \not\prec_{ch} x)$$

Now we can define the uniqueness of the root as follows:

$$\varphi_{unique\ root} = \exists x(\varphi_{no\ child}(x) \wedge \forall y(\varphi_{no\ child}(y) \rightarrow y = x))$$

Another condition that a JSON document must satisfy is the uniqueness of the keywords, in other words, if a node has more than one child, those children must have different labels. The following formula captures this property:

$$\varphi_{unique\ keywords} = \forall x \forall y \forall z(x \prec_{ch} y \wedge x \prec_{ch} z \wedge y \neq z \rightarrow \lambda(y) \neq \lambda(z))$$

Additionally, we need to force every non-root property to have a keyword(i.e.λ cannot be a partial function on keywords). This can be done by checking the conformance to the following formula:

$$\varphi_{nonempty\ keywords} = \forall x(\neg P_{root}(x) \wedge \exists y(x \prec_{ch} y) \rightarrow \exists s(\lambda(x) = s))$$

Moreover, if we look carefully at our encoding of JSON documents, it is easy to note that leaf nodes don't have siblings. Consider the following formula to encode leafs:

$$\varphi_{leaf}(x) = \forall y(x \not\prec_{ch} y)$$

We encode the property described above as follows

$$\varphi_{no\ sibling} = \forall x \forall y \forall z(x \prec_{ch} y \wedge x \prec_{ch} z \wedge (\varphi_{leaf}(y) \vee \varphi_{leaf}(z)) \rightarrow y = z)$$

Finally we give a formula φ_{JSON} that every JSON document must satisfy

$$\varphi_{\text{JSON}} = \varphi_{unique\ root} \wedge \varphi_{unique\ keywords} \wedge \varphi_{nonempty\ keywords} \wedge \varphi_{no\ sibling}$$

Now we proceed to construct a formula for each schema, we do this inductively on the syntax of JSON Schema. It is important to mention that we provide a translation from the core fragment of JSON Schema. The remaining keywords can be simulated by combining the core operators. For example, we do not consider numeric restrictions, since `multipleOf`, `minimum` and

Note that if we threat $\varphi_{unique\ root}$ as a free variable formula it defines P_{root}.

`maximum` can be easy simulated by the use of a regular expression (Sakarovitch, 2009). Moreover, the `enum` keyword can be simulated for both strings and objects. In the case of strings, we just need to force the string value by the use a expression. For objects, we can always provide a schema that define a single object, just by combining the `required`, `properties` and `additionalProperties` keywords.

Let us start out construction with string schemas, consider a string schema S

```
{
 "type": "string",
 "pattern": "L"
 }
```

where L is a regular expression over Σ. At this point, one can devise a very simple formula in MSO, let φ_S be this formula

$$\varphi_S(x) := \exists y \forall z (x \prec_{ch} y \wedge y \not\prec_{ch} z \wedge \exists s(\lambda(y) = s \wedge R_L(s))$$

Here the first part of the formula is asserting that x has a child y that must be a leaf node and the second one forces it to have a string value in L. Note that in the case that we don't have the presence of `pattern` we just have to remove the last restriction. It is important to mention that x must be a free variable since we are working inductively in the syntax of JSON Schema(i.e. S is not necessary at the topmost level of the document).

In the case of objects, the formula becomes more complicated, consider the following object schema S

```
{
   "type": "object",
   "required": ["s_1",...,"s_n"],
   "properties": {
              "k_1": S_1,
              ...
              "k_m": S_m
              },
   "patternProperties": {
              "r_1": D_1,
```

```
                ...
                "r_l": D_l
                },
"additionalProperties": A
}
```

Intuitively we have to use our inductive step to assign a formula for each subschema present in the definition of S. Let $\varphi_{S_1}, ..., \varphi_{S_m}, \varphi_{D_1}, ..., \varphi_{D_l}, \varphi_A$ be those formulas, we can construct φ_S as follows

$$\varphi_S(x) :=$$

$$\exists y_1 \ldots \exists y_n((x \prec_{ch} y_1 \wedge \lambda(y_1) = s_1) \wedge \ldots \wedge (x \prec_{ch} y_n \wedge \lambda(y_n) = s_n))$$

$$\wedge \forall z_1(x \prec_{ch} z_1 \wedge \lambda(z_1) = k_1 \rightarrow \varphi_{S_1}(z_1)) \wedge \ldots \wedge \forall z_m(x \prec_{ch} z_m \wedge \lambda(z_m) = k_m \rightarrow \varphi_{S_m}(z_m))$$

$$\wedge \forall u_1(x \prec_{ch} u_1 \wedge R_{r_1}(\lambda(u_1)) \rightarrow \varphi_{D_1}(u_1)) \wedge \ldots \wedge \forall u_l(x \prec_{ch} u_l \wedge R_{r_l}(\lambda(u_l)) \rightarrow \varphi_{D_l}(u_l))$$

$$\wedge \forall v(x \prec_{ch} v \wedge \lambda(v) \neq k_1 \wedge \ldots \wedge \lambda(v) \neq k_m \wedge \neg R_{r_1}(\lambda(v)) \wedge \ldots \wedge \neg R_{r_l}(\lambda(v)) \rightarrow \varphi_A(v))$$

The first line asserts that the keywords in the required section must be present in the children of x. The second and third line code the `properties` and `patternProperties` restrictions respectively. Finally, the last line translates the `additionalProperties` restriction. Similarly as the string case, if any of the restrictions is not present in the schema, we just remove one of the lines of the formula. In the case that S contains `additionalProperties:false`, we can simply permute the last line by

$$\forall v(\lambda(v) \neq k_1 \wedge \ldots \wedge \lambda(v) \neq k_m \wedge \neg R_{r_1}(\lambda(v)) \wedge \ldots \wedge \neg R_{r_l}(\lambda(v)) \rightarrow x \not\prec_{ch} v)$$

We continue by showing how to construct the formula in the presence of single enumeration and boolean combinations. To illustrate this step consider the following schemas:

$S_{allOf} =$ {"allOf": [$S_1, \ldots, S_n$]} $\qquad$ $S_{anyOf} =$ {"anyOf": [$S_1, \ldots, S_n$]}

$S_{not} =$ {"not": S'} $\qquad$ $S_{enum} =$ {"enum": [$J_1, \ldots, J_n$]}

We can easily deduce a formula for each of these schemas by taking advantage advantage of the essence of boolean logic. Let $\varphi_{allOf}, \varphi_{anyOf}, \varphi_{not}, \varphi_{enum}$ be the formulas for the schemas above, from the inductive step we obtain the following construction:

$$\varphi_{allOf}(x) := \varphi_{S_1}(x) \wedge \ldots \wedge \varphi_{S_n}(x)$$
$$\varphi_{anyOf}(x) := \varphi_{S_1}(x) \vee \ldots \vee \varphi_{S_n}(x)$$
$$\varphi_{not}(x) := \neg\varphi'_S(x)$$

where $\varphi'_S, \varphi_{S_1}, \ldots, \varphi_{S_n}$ are the formulas assigned to the corresponding subschemas.

Finally, in order to provide a full mapping from JSON Schema to MSO we must give a construction of both the root of the document and the definitions section. Consider the following JSON Schema document JS:

```
{
   "definitions": {
                  "d_1": S_1,
                  ...
                  "d_n": S_n
                           },
        S
 }
```

Here we take advantage of MSO by assigning a predicate variable D_i for each definition d_i for $1 \leq i \leq n$. Let φ_{JS} be the formula for the entire document, we can build it as follows:

$$\varphi_{JS} := \varphi_{JSON} \wedge \exists x \exists D_1 \ldots \exists D_n (P_{root}(x) \wedge \varphi_S(x) \wedge \forall y((D_1(y) \rightarrow \varphi_{S_1}(y)) \wedge \ldots \wedge (D_n(y) \rightarrow \varphi_{S_n}(y)))$$

Furthermore, we can derive the construction of the formula for reference schemas directly from above. Let $S = \{\texttt{"\$ref":"\#/definitions/}d_i\texttt{"}\}$, we can construct φ_S as follows

$$\varphi_S(x) := D_i(x)$$

Note that we don't need free variables in the definition of φ_{JS}, since it constrains the whole document. Finally, for each JSON document J and JSON Schema document JS by simple construction we obtain that there is a formula φ_{JS} such that

$$J \models JS \text{ if and only if } T(J) \models \varphi_{JS}$$

where $T(J)$ is the encoding of J as a MSO structure. □

C.3. Proof of Proposition 3

PROOF. As we already got the upper bound for JSCHVALIDATION(J, S) from Proposition 2, we proceed to establish a proper lower bound for the problem. In order to achieve this, we perform a LOGSPACE reduction from Monotone Circuit Value problem, which is known to be PTIME-complete (Goldschlager, 1977). Let C be a monotone circuit circuit with its input coded as x_i for some $i \geq 0$ and τ its corresponding valuation. We can generate a schema S by executing a simple procedure over (C, τ).

We start by adding one definition for each input x_i, the definition forces the value to be exactly the one assigned by τ. Then, we traverse the circuit in a depth-first fashion, starting from the root. If we encounter an `AND` gate, we add an `allOf` restriction to the schema, with a number of subschemas equal to the number of inputs the gate has. If the gate is an `OR` we add an `anyOf` restriction. In both cases we repeat the process until we reach the leafs of C. At this point, we simply add a reference to the definition assigned to the input. The detailed procedure is shown at Algorithm 5.

Algorithm 5 Reduction from Monotone Circuit.

procedure CIRCUITTOSCHEMA(C, τ)
  Let S be an empty JSON Schema document
  for x_i in Inputs(C) **do**
    if $\tau(x_i) ==$ TRUE **then**
      S["definitions"].append("x_i":{"enum":[true]})
    else if $\tau(x_i) ==$ TRUE **then**
      S["definitions"].append("x_i":{"enum":[false]})
    end if
  end for
  Let v be $Root(C)$
  S.body.append(TRAVERSECIRCUIT(v))
  return S

procedure TRAVERSECIRCUIT(v)
  if $v == \wedge(v_1, ..., v_n)$ **then**
    return {"allOf":[TRAVERSECIRCUIT(v_1),...,TRAVERSECIRCUIT(v_n)]}
  else if $v == \vee(v_1, ..., v_n)$ **then**
    return {"anyOf":[TRAVERSECIRCUIT(v_1),...,TRAVERSECIRCUIT(v_n)]}
  else if $v == x_i$ **then**
    return {"$ref": "#/definitions/$x_i$"}
  end if

Finally, our JSON document J just contains the value `true`. It is straightforward to prove that $J \models S$ if and only if $\tau(C) = \texttt{true}$. However, it is important to mention that the procedure works on LOGSPACE, since we do not need to have the whole circuit in memory during the traversal (DFA keeps a stack with logarithmic memory).

□

C.4. Proof of Lemma 1

PROOF. Given a schema S,we proceed by structural induction over the depth of the accepting documents of S, let us consider the following predicate:

$P(h)$: For every satisfiable schema that just accepts documents of at least depth h, there exists a polynomial size document that conforms to it.

Base Case

As our base case we start by showing that $P(0)$ is true. Let S be a satisfiable schema of depth 0, we can see S at its topmost level as a boolean combination of schemas. It is well known that every boolean combination can be equivalently written as a formula in disjunctive normal form (DNF). We now rewrite S in its equivalent DNF schema as follows:

$$S \equiv \bigvee_{i} C_i$$

Here each C_i is a conjunctive clause of schemas and negation of schemas. Since S is satisfiable, it is easy to devise that at least one C_i must be satisfiable. Let C be this satisfiable clause, as C is a conjunctive clause of schemas it must have the following structure:

$$C = \bigwedge_{j=1}^{n} L_j \tag{C.1}$$

Here each L_j is a schema or a negation of a schema that defines one or more JSON types, for $1 \leq j \leq n$. As C is satisfiable we know that there is a document that conforms to it. Then, all L_j's must agree in at least one accepting type of document. Now we procede by showing that for each type, C presents a polynomial size document that conforms to it.

Strings

Let us asume that C is satisfiable by a string document and by consequence every L_j also conforms to it. Now each L_j can only be one of the following type of schemas:

- string schema

According to our grammar we can asume there is a combination `allOf`, `anyOf` and `not` keywords in the top layer of every schema.
Strings, numerics, booleans, null, arrays and objects.

- negation of a string schema
- negation of a numeric schema
- negation of a boolean schema
- negation of a null schema
- negation of an array schema
- negation of an object schema

It is clear that in the last five cases L_j accepts every string, so we just focus on both strings schemas and negated string schemas:

$$L_j : \{\texttt{minLength:}w_j\texttt{, maxLength:}l_j\} \mid \{\texttt{not:}\{\texttt{minLength:}w_j\texttt{, maxLength:}l_j\}\}$$

We can see these restrictions as natural numbers inequalities, let $|x|$ be the length of a string:

$$L_j : (|x| \geq w_j \wedge |x| \leq |l_j|) \mid (|x| \leq w_j \vee |x| \geq l_j)$$

Combining with C.1 we can see C as boolean combination of inequalities:

$$\left(\bigwedge_{L_j \text{is not negated}} (|x| \geq w_j \wedge |x| \leq l_j)\right) \wedge \left(\bigwedge_{L_j \text{is negated}} (|x| \leq w_j \vee |x| \geq l_j)\right)$$

It is well know that we can transform these boolean combinations of inequalities to DNF as follows:

$$C \equiv \bigvee_k I_k$$

Here each I_k is a conjunctive clause of inequialities. We can see each I_k as a (possibly) closed interval of natural numbers. Furthermore, as C is satisfiable there must exists a non-empty interval $I_{k'}$:

$$I_{k'} : (w' \leq |x| \leq l')$$

Here w', l' must appear in one of the restrictions of C. Finally, as we are looking for an upper bound we can just take any string document J such that:

$$I_{k'} : (w' \leq |J| \leq l')$$

Clearly $|J|$ is bounded by l', we conclude that $|J|$ can be taken as a polynomial size witness for S.

Remark: It is possible that $I_{k'}$ cold be an open interval. In this case we can just take any string of length w' as our witness.

Numeric Types

In the case of numeric types, since integer values are more restrictive than rational numbers, we just give a proof for the integer case. However, one can prove the same result relaxing the problem for rational numbers.

Let us asume that C is satisfiable by an integer document and by consequence every L_j must accept an integer document. Now each L_j can only be one of the following type of schemas:

- integer schema
- negation of an integer schema
- negation of a string schema
- negation of a boolean schema
- negation of a null schema
- negation of an array schema
- negation of an object schema

Similarly as the string case, we just focus on both integer schemas and negated integer schemas, now L_j can be seen as:

$$\{\texttt{minimum:}w_j\texttt{,maximum:}l_j\texttt{,mutipleOf:}r_j\}|\{\texttt{not:}\{\texttt{minimum:}w_j\texttt{,maximum:}l_j\texttt{,mutipleOf:}r_j\}\}$$

First of all, note that we can avoid the `exclusiveMinimum` and `exclusiveMaximum` keywords by simply adding and subtracting 1 from w_j and l_j respectively. Now we can handle this inequalities using a similar procedure as the one used with string schemas. The only difference arises in the fact that we can only choose numbers that satisfy the `multipleOf` restrictions.

$$L_j : (x \geq w_j \wedge x \leq l_j \wedge x = r_j c_j) \mid (x \leq w_j \vee x \geq l_j \vee x \neq r_j c_j) \qquad c_j \in \mathbb{Z}$$

Combining with C.1 we can see C as boolean combination of inequalities:

$$\left(\bigwedge_{L_j \text{is not negated}} (x \geq w_j \wedge x \leq l_j \wedge x = r_j c_j) \right) \wedge \left(\bigwedge_{L_j \text{is negated}} (x \leq w_j \vee x \geq l_j \vee x \neq r_j c_j) \right)$$

Now we rewrite C in its corresponding DNF:

$$C \equiv \bigvee_{k} I_k \tag{C.2}$$

Here each I_k is a conjunctive clause of inequialities and multiplicities. In other words, we can see each I_k as a (possibly) closed interval of common multiple numbers. Based on our hypothesis, as C is satisfiable there must exists a non-empty interval $I_{k'}$:

$$I_{k'} : (w \leq x \leq l) \wedge \left(\bigwedge_{L_{k''k''} \text{ is a positive schema}} (x = r_{k''}c_{k''}) \right) \wedge \left(\bigwedge_{L_{k''k''} \text{ is a negative schema}} (x \neq r_{k''}c_{k''}) \right) \tag{C.3}$$

Let J be the integer document that satisfies all the restrictions above. Now we have two cases, on one hand if the interval is closed it must be bounded by one of the restrictions in it, and by consequence J is bounded. On the other hand, if $I_{k'}$ is not bounded we can choose the first number that satisfies the multiplicity restrictions. The size of this number cannot be bigger than

$$w + LCM_{k''}(r_{k''}) \tag{C.4}$$

Here $LCM_{k''}(r_{k''})$ is the least common multiple between every $r_{k''}$. Clearly J is bounded by the size of the `multipleOf` restrictions, and by consequence we can take it as a polynomial size witness for S.

Remark 1: It is important to mention that the `multipleOf` restrictions that come from negative schemas are not taken in account in the final step. There cannot be the case that they match with the multiples presented in the non-negated schemas, since the schema would not be satisfiable.

Remark 2: In the case that the interval is open to the negative infinity, we can do the analogous step by taking l instead of w and substracting the LCM.

Null and Boolean Types

Since the only values that they can take are `null`, `true` and `false`, the size of the documents is always constant independently of the boolean restrictions on the top layer of the schemas.

Here we asume that both the restriction and the witness are binary encoded.

Object and Array Types

As we know that C accepts a document of depth 0, the only possible object or array is the empty instance of them. Now we can always take those instances as our polynomial size witnesses.

Inductive Hypothesis

As we mentioned before our inductive hypothesis is

$P(h)$: For every satisfiable schema that just accepts documents of at least depth h, there exists a polynomial size document that conforms to it.

Let us assume that $P(h)$ is true for all h greater than 0.

Inductive Thesis

Now we procede to prove that $P(h+1)$ is true. Let S be a satisfiable JSON Schema that accepts a document of depth $h+1$. Similarly as in the base case we procede to rewrite S in its equivalent DNF schema as follows:

$$S \equiv \bigvee_i C_i \tag{C.5}$$

Here each C_i is a conjunctive clause of both schemas and negation of schemas. Since S is satisfiable it is easy to devise that at least one C_i must be satisfiable. Let C be this satisfiable clause, as C is a conjunctive clause of schemas it must have the following structure:

$$C = \bigwedge_{j=1}^{n} L_j \tag{C.6}$$

Here each L_j is a schema or negated schema that defines one or more JSON types, for $1 \leq j \leq n$. As C is satisfiable we know that there is a document that conforms to it. Then, all L_j's must agree in at least one accepting type of document. As we know that the depth of this document must be at least $h+1$, then it can be either an object or an array. Now we procede by showing that for each of these types that C presents a polynomial size document that conforms to it.

Objects

Let us asume that C is satisfiable by an object document and by consequence every L_j must accept an object document. Now each L_j can only be one of the following type of schemas:

- object schema

- negation of an object schema
- negation of a string schema
- negation of a boolean schema
- negation of a null schema
- negation of an array schema
- negation of an integer schema

It is clear that in the last five cases L_j will accept every object, so we just focus on both object schemas and negated object schemas:

$\{$`required:`$[key]_j$`, properties:`$\{key$`:` $S_{j,key}\}_j$`, additionalPropreties:`$S_{j,adPr}\}$

$\{$`not:{required:`$[key]_j$`, properties:`$\{key$`:` $S_{j,key}\}_j$`, additionalPropreties:`$S_{j,adPr}\}\}$

Here $[key_k]_j$ is the list of required keywords of the L_j, $\{key_k : S_{j,key_k}\}_j$ is the list of properties of L_j with its respective schema and $S_{j,adPr}$ is the `additionalProperties` schema for L_j.

Now we look for an expression that formally describes the constraints that C must satisfy. Based on our hypothesis, we know that C accepts an object document that satisfies every L_j. Let J be this document, we start by checking the existence of the required keywords for each L_j, the notation we use is the same defined in Section 2.1:

$$\varphi_{req}(L_j) = \bigwedge_{key\ in\ L_j[\text{required}]} (J[key] \neq \varnothing)$$

Besides, we need to check that each keyword in J must conform to it's corresponding schema in the properties section of each L_j:

$$\varphi_{prop}(L_j) = \bigwedge_{key'\ in\ J} \left(\bigwedge_{L_j[\text{properties}][key']\neq\varnothing} J[key'] \models L_j[\text{properties}][key'] \right)$$

Similarly we need to check the additional properties section:

$$\varphi_{addProp}(L_j) = \bigwedge_{key'\ in\ J} \left(\bigwedge_{L_j[\text{additionalProperties}][key']=\varnothing} J[key] \models L_j[\text{additionalProperties}] \right)$$

In the case of negated schemas we can just negate the formulas above as follows:

$$\varphi_{\neg req}(L_j) = \bigvee_{key\ in\ L_j[\text{required}]} (J[key] = \varnothing)$$

$$\varphi_{\neg prop}(L_j) = \bigvee_{key'\ in\ J} \left(\bigvee_{L_j[\text{properties}][key'] \neq \varnothing} J[key'] \models \neg L_j[\text{properties}][key'] \right)$$

$$\varphi_{\neg addProp}(L_j) = \bigvee_{key'\ in\ J} \left(\bigvee_{L_j[\text{additionalProperties}][key'] = \varnothing} J[key] \models \neg L_j[\text{additionalProperties}] \right)$$

Note that when we negate a validation restriction in the form of $J \models S$, we just need to complement the schema with the `not` keyword, here the notation used is $J \models \neg S$. Finally, we can rewrite the restrictions in C as follows:

$$\psi : \bigwedge_{L_j \text{is not negated}} (\varphi_{req}(L_j) \wedge \varphi_{prop}(L_j) \wedge \varphi_{addProp}(L_j)) \wedge \bigwedge_{L_j \text{is negated}} (\varphi_{\neg req}(L_j) \vee \varphi_{\neg prop}(L_j) \vee \varphi_{\neg addProp}(L_j))$$

It is clear that we can rewrite ψ in its equivalent DNF formula as follows:

$$\psi \equiv \bigvee_{i'} R_{i'}$$

Here each $R_{i'}$ is a conjunctive clause of required, not required and satisfiability restrictions:

$$R_{i'} = \bigwedge J[key] \neq \varnothing \wedge \bigwedge J[key] = \varnothing \wedge \bigwedge J[key] \models S'$$

Finally as we know that ψ is satisfiable, we know that there is a $R_{i''}$ that has a model that conforms to it. Since we know that every S' accepts a document of height h we can apply our inductive hypothesis and conclude that C and therefore S has a number of polynomial documents equal to the number of $J[key] \neq \varnothing$ restrictions in $R_{i''}$ hanging from the root. We conclude by an inductive argument that $P(h+1)$ holds for object schemas.

Arrays

Let us asume that C is satisfiable by an array document and by consequence every L_j must accept an array document. Now each L_j can only be one of the following type of schemas:

- array schema
- negation of an array schema
- negation of a string schema
- negation of a boolean schema
- negation of a null schema
- negation of an object schema
- negation of an integer schema

It is clear that in the last five cases L_j will accept every array, so we just focus on both array schemas and negated array schemas:

$\{$`items:` S'`, minItems:` w_j`, maxItems:` l_j`, uniqueItems: true`$\}$
$\{$`not:{items:` S'`, minItems:` w_j`, maxItems:` l_j`, uniqueItems: true`$\}\}$

For the sake of simplicity, we do not consider the items restriction in the form of `items:` $[S_1, ..., S_n]$. Moreover, we assume that the `"minItems"` and `"maxItems"` keyword values are coded in unary. The reason of this is the fact that a decimal encoding could lead to a exponential blow-up in the number of items. Regardless, the prove can be adapted to include these constrains. Now we look for an expression that formally describes the restrictions that C must satisfy. Based on our hypothesis, we know that C accepts an array document that satisfies every L_j. Let J be this document and $|\cdot|$ be the length of the array, we start by defining the constrains for the non-negated schemas:

$$\varphi_1(L_j) = w_j \leq |J| \leq l_j \wedge \bigwedge_{k=1}^{|J|} \left(J[k] \models S'\right) \wedge \bigwedge_{i'=1}^{|J|} \left(\bigwedge_{j'=1 \wedge j' \neq i'}^{|J|} \left(J[i'] \neq J[j']\right) \right)$$

Again, we use the notation defined in Section 2.1 to retrieve the value of the elements of J. Now we give a proper formula for the negated schemas:

$$\varphi_2(L_j) = w_j > |J| > l_j \vee \bigvee_{k=1}^{|J|} \left(J[k] \models \neg S'\right) \wedge \bigvee_{i'=1}^{|J|} \left(\bigvee_{j'=1 \wedge j' \neq i'}^{|J|} \left(J[i'] = J[j']\right) \right)$$

Similarly as the object case, instead of negating the model constrain, we negate the schema by nesting it into a `not` restriction. Finally, we can rewrite the restrictions in C as follows:

$$\psi : \bigwedge_{L_j \text{is not negated}} (\varphi_1(L_j)) \wedge \bigwedge_{L_j \text{is negated}} (\varphi_2(L_j))$$

It is clear that we can rewrite ψ in its equivalent DNF formula as follows:

$$\psi \equiv \bigvee_{i'} R_{i'}$$

Here each $R_{i'}$ is a conjunctive clause of satisfiability, equality, inequality and length restrictions:

$$R_{i'} = \bigwedge (J[k] \models S') \wedge \bigwedge (J[i'] = J[j']) \wedge \bigwedge (J[i'] \neq J[j']) \wedge \bigwedge (w_j \leq |J| \leq l_j)$$

Finally as we know that ψ is satisfiable, we know that there is a $R_{i''}$ that has a model that conforms to it. Since we know that every S' accepts a document of height h we can apply our inductive hypothesis and conclude that C and therefore S has a number of polynomial documents shorter than l_j hanging from the root. Moreover, the equality constrains do not affect the size of the witness. In the case that there is no l_j present in the schemas we can just give a document with the minimum w_j as witness. It is important to mention that we did not consider the single enumeration case. The reason lays in the fact that we can easily take as witness the value that the enumeration forces. We conclude by an inductive argument that $P(h+1)$ holds for array schemas. $\square$

C.5. Proof of Theorem 2

PROOF. We proceed inductively over the syntax of JSON Schema. Here assume that all trees come from the encoding of a valid JSON document. Before providing the construction of the QATA we define two states to deal with the acceptance of the schemas.

- $q_\top$: Used to spread an accepting state through the tree.
- $q_\bot$: Used to spread a rejecting state through the tree.

Let S be an schema, the way we perform the induction is by generating a formula $\varphi(S)$ for the right side of the rule associated with S. Additionally, in each case we add a set of rules $\Delta(S)$ to the rules of the automaton.

We start by showing how to construct a QATA from the string case. Let S be a string schema:

```
{
 "type": "string",
 "pattern": "La"
 }
```

Here we can construct $\varphi(S)$ as follows:

$$\varphi(S) = \exists q_L$$

Additionally we add two rules to check the compliance to the regular expression:

$$q_L(L) \to true$$

$$q_L(L^c) \to false$$

Note that we use the L^c to represent every string that is not L (i.e. the complement). Besides, in the case that the pattern restriction is missing we can just assume it appears with $L = \Sigma^*$. Finally, it is important to mention that we translate every regular expression to an alternating word automaton. In this way, we avoid an exponential blow up from the complementation. Regardless of this, for the ease of the exposition we show them as regular expressions.

A valid JSON document is a document that satisfies the restrictions established in the demonstration of Theorem 1.

Now we give the construction for object schemas. As the `properties` keyword is just a particular case of `patternProperties`, we just provide the construction for pattern case. Let S be an object schema

{
  "type": "object",
  "required": ["s_1",...,"s_n"],
  "patternProperties": {
            "r_1": D_1,
            ...
            "r_m": D_m
            },
  "additionalProperties": A
  }

Here we can construct $\varphi(S)$ as follows:

$$\varphi(S) = \exists q_{s_1} \wedge \ldots \wedge \exists q_{s_n} \wedge \forall q_{r_1} \wedge \ldots \wedge \forall q_{r_m,} \wedge \forall q_A$$

Additionally, we have to add a set of rules $\Delta(S)$ to the automaton. The rules bellow allow us to state that the required keywords must be present in the object:

$$q_{s_1}(s_1) \rightarrow \forall q_\top$$

$$q_{s_1}(s_1^c) \rightarrow \exists q_\bot$$

$$\ldots$$

$$q_{s_n}(s_n) \rightarrow \forall q_\top$$

$$q_{s_n}(s_n^c) \rightarrow \exists q_\bot$$

Now give the rules we need to satisfy the `patternProperties` section:

$$q_{r_1}(r_1) \to \varphi_{D_1}$$
$$q_{r_1}(r_1^c) \to \forall q_\top$$
$$\dots$$
$$q_{r_m}(r_m) \to \varphi_{D_m}$$
$$q_{r_m}(r_m^c) \to \forall q_\top$$

In this case we are treating with an universal quantifier just to deal with the implication that the `patternProperties` is validated. In other words, if a keyword is in the language of one of the regular expressions, then it must satisfy the formula assigned to that language. In other case, the state checking that expression accepts the word and spread the accepting state to the leafs. Again, every regular expression is previously encoded into an alternating word automaton to avoid an exponential blow up.

Finally we add a rule for checking the `additionalProperties` section:

$$q_A\left(\bigcap_{i=1}^{m} r_i^c\right) \to \varphi_A$$
$$q_A\left(\bigcup_{i=1}^{m} r_i\right) \to \forall q_\top$$

We continue by showing how to construct the automaton in the presence of single enumeration and boolean combinations. To illustrate this step consider the following schemas:

$$S_{allOf} = \{\texttt{"allOf": [}S_1\texttt{,...,}S_n\texttt{]}\}$$
$$S_{anyOf} = \{\texttt{"anyOf": [}S_1\texttt{,...,}S_n\texttt{]}\}$$
$$S_{not} = \{\texttt{"not": }S'\}$$

We can easily deduce a formula for each of these schemas by taking advantage advantage of the essence of boolean logic. Let $\varphi(S_{allOf}), \varphi(S_{anyOf}), \varphi(S_{not})$ be the formulas for the schemas above, from the inductive step we obtain the following construction:

$$\varphi(S_{allOf}) := \varphi(S_1) \wedge \ldots \wedge \varphi(S_n)$$
$$\varphi(S_{anyOf}) := \varphi(S_1) \vee \ldots \vee \varphi(S_n)$$
$$\varphi(S_{not}) := \overline{\varphi(S')}$$

where $\varphi(S'), \varphi(S_1), \ldots, \varphi(S_n)$ are the formulas assigned to the corresponding subschemas. Besides, in the `"not"` case we just complement the formula of the subschema. Note that this can be easily done by exchanging the accepting and rejecting states and negating the quantifiers/boolean operators.

Similar to the other cases we still need to add some rules to Δ. In the case of the `"anyOf"` and the `"allOf"` schemas, we simply add $\Delta(S_i)$ for $i \leq n$ to $\Delta(S_{allOf})$ or $\Delta(S_{anyOf})$. Despite of this, in the case of the negation we need to recursively complement all the formulas in $\Delta(S')$ in a similar way as we complemented $\varphi(S')$.

It is important to mention that all the complementations can be performed in polinomial time, similar as the alternating tree automaton models presented in (Comon et al., 2007). This is done by performing a single iteration through the automaton complementing each rule. In each rule we exchange the universal and existential quantifiers, and then apply the boolean complementation for each logical connective. Besides, we need exchange the accepting and rejecting states.

Finally, in order to provide a full mapping from JSON Schema to QATA we must give a construction of both the root of the document and the definitions section. Consider the following JSON Schema document JS:

{
"definitions": {
"d_1": S_1,
...
"d_n": S_n
},
S

$\}$

Here we take advantage of the inductive definition and for each definition d_i we add $\Delta(S_i)$ to the rules of our automaton. Furthermore, we can derive the construction of the formula for reference schemas directly from above. Let $S = \{\texttt{"\$ref":"\#/definitions/}d_i\texttt{"}\}$, we can construct $\varphi(S)$ as follows

$$\varphi(S) := \varphi(S_i)$$

Finally, given a JSON Schema document JS with schema S in the body, we can construct an automaton $\mathcal{A}_{JS}(Q, \Sigma, \mathcal{I}, \Delta)$ by taking $\Delta = \Delta(S) \cup R_0$. Where the rules in R_0 are defined as follows:

$$q_0(\epsilon) \to \varphi(S)$$
$$q_\top(\Sigma^*) \to \forall q_\top$$
$$q_\top(\Sigma^*) \to true$$
$$q_\bot(\Sigma^*) \to \exists q_\bot$$
$$q_\bot(\Sigma^*) \to false$$

In the first rule we define the initial state to start the run from the top, where $q_0 \in \mathcal{I}$. In the other rules, we take advantage of the nondeterminism to spread the accepting and rejecting states across the branches of the tree.

To conclude the proof, we appeal to the intuition to state that for every valid JSON document J it must be true that

$$J \models JS \text{ if and only if } T(J) \in L(\mathcal{A}_{JS})$$

where $T(J)$ is the tree codification of J. $\square$

C.6. Proof of Theorem 3

PROOF. Recall we need to prove that given a *recursive* schema S, it is possible to check that S it is satisfiable in $O(2^{|S|})$. However, from Theorem 2 we know it is possible to give a translation from JSON Schema to QATA in polynomial time. Because of this, we just need to provide an algorithm to compute emptiness of QATA in exponential time to achieve our desired result. In this context, an emptiness algorithm is much more intuitive when it is run bottom-up fashion, in comparison to it's top-down version. For this reason, we provide an algorithm designed to be run over a bottom-up *quantified alternating tree automata*.

Definition 6 (Bottom up quantified alternating tree automata)**.** A bottom-up **quantified alternating tree automata** (QATA) $\mathcal{A}$ over Σ is a tuple $(Q, \Sigma, \mathcal{F}, \Delta)$ where Q is a set of states, $\mathcal{F} \subseteq Q$ is a set of final states and Δ is a finite set of transition rules of the following type:

$$R_\Sigma(\varphi) \to q$$

where $R_\Sigma \subseteq \Sigma^*$ is a regular language over Σ and $\varphi \in \mathcal{B}^+_{\exists\forall}(Q)$. Here $\mathcal{B}^+_{\exists\forall}(Q)$ is the set of propositional formulas over Q, where each state $q \in Q$ is preceded by a quantifier. Again, the regular languages are coded into alternating word automata and showed as regular expressions for readability reasons.

Let v be a node labeled s with n children. A run of $\mathcal{A}$ over a tree structure $\mathfrak{T}$ is a mapping $\rho : T_D \to 2^Q$ such that if $\rho(v) = S$, then for each state q_S in S it must be true that:

$$\text{if } s(\psi) \to q_S \text{ then } \{\rho(u_1), ..., \rho(u_n)\} \vDash \psi \text{ where } s \prec_{ch} u_i \text{ for every } 1 \leq i \leq n.$$

Here the notion of satisfaction is the same as the one define for the top-down version. A run ρ is successful if $q \in \rho(\epsilon)$ for some final state $q \in \mathcal{F}$. Again, the definition corresponds to a nondeterministic model, where a string can match several rules in the left side of the transition relation.

Similar to classic tree automata, nondeterministic QATA can be run in both directions without loosing expressive power. The following proposition formalises this statement.

LEMMA 2. *A language is accepted by a* nondeterministic top-down *QATA if and only if it is accepted by a* nondeterministic bottom-up *QATA*.

PROOF. The proof is left to the reader. **Hint.** Reverse the arrows and exchange the sets of initial and final states. □

With this last result we can finally start to devise our emptiness algorithm. In general terms, our algorithm attempts to perform a determinization of the automaton's rules and then compute all the reachable sets of states, until we reach a set containing a final state.

Let us start with the determinization, consider the following set of rules Δ=

$$R_1(\varphi_1) \to q_1$$

$$\dots$$

$$R_n(\varphi_n) \to q_n$$

The pseudocode of the determinization procedure for Δ is shown in Algorithm 6.

Algorithm 6 Determinization algorithm for QATA.

1: **procedure** DETERMINIZE(Δ)
2: $\quad\Delta^D = \varnothing$
3: $\quad$**for** $C \subseteq \Delta$ **do**
4: $\quad\quad\Delta^D \leftarrow \bigcap_{R_i \in C} R_i \left(\bigwedge_{\varphi_i \in C} \varphi_i \wedge \bigwedge_{\varphi_i \notin C} \neg\varphi_i \right) \to \bigcup_{q \in C} \{q\}$
5: $\quad\quad\Delta^D \leftarrow \bigcap_{R_i \in C} R_i \cap \bigcap_{R_i \notin C} R_i^c \left(\bigwedge_{\varphi_i \in C} \varphi_i \right) \to \bigcup_{q \in C} \{q\}$
6: $\quad$**end for**
7: $\quad$**return** Δ^D

The procedure described in Algorithm 6 checks every subset of rules and then construct a pair of rules such that given a pair $(w, S) \in \Sigma^* \times 2^Q$ it must match at most one of the rules in Δ^D. Moreover, our algorithm also computes all the reachable states from (w, S) and sets them in the right side of the rule. This is the key step that is going to help us when we perform the emptiness algorithm. The following Lemma helps us to confirm the correctness of our determinization:

LEMMA 3. *Let Δ be a set of rules and $\Delta^D = \text{DETERMINIZE}(\Delta)$, then Δ^D is deterministic.*

PROOF. By contradiction, suppose there is a pair $(w, S) \in \Sigma^* \times 2^Q$ and two rules $D_1, D_2 \in \Delta^D$ such that $(w, S) \models D_1$ and $(w, S) \models D_2$. Now we proceed by cases depending on the form of each rule.

(i)

$$D_1 \text{ is of the form } \bigcap_{R_i \in C_1} R_i \left(\bigwedge_{\varphi_i \in C_1} \varphi_i \wedge \bigwedge_{\varphi_i \notin C_1} \neg\varphi_i \right) \rightarrow \bigcup_{q \in C_1} \{q\}.$$
$$D_2 \text{ is of the form } \bigcap_{R_i \in C_2} R_i \left(\bigwedge_{\varphi_i \in C_2} \varphi_i \wedge \bigwedge_{\varphi_i \notin C_2} \neg\varphi_i \right) \rightarrow \bigcup_{q \in C_2} \{q\}.$$

As D_1 has the same form of D_2 it must be true that $C_1 \neq C_2$. Without loss of generality, there must be a rule (R, φ, q) such that $(R, \varphi, q) \in C_1$ and $(R, \varphi, q) \notin C_2$. Then, by the form of D_2 it must be true that $S \models \neg\varphi \Rightarrow S \not\models \varphi$. Finally, as D_1 includes φ we have that $(w, S) \not\models D_1$, which contradicts our initial supposition.

(ii)

$$D_1 \text{ is of the form } \bigcap_{R_i \in C_1} R_i \cap \bigcap_{R_i \notin C_1} R_i^c \left(\bigwedge_{\varphi_i \in C_1} \varphi_i \right) \rightarrow \bigcup_{q \in C_1} \{q\}.$$
$$D_2 \text{ is of the form } \bigcap_{R_i \in C_2} R_i \cap \bigcap_{R_i \notin C_2} R_i^c \left(\bigwedge_{\varphi_i \in C_2} \varphi_i \right) \rightarrow \bigcup_{q \in C_2} \{q\}.$$

As D_1 has the same form of D_2 it must be true that $C_1 \neq C_2$. Without loss of generality, there must be a rule (R, φ, q) such that $(R, \varphi, q) \in C_1$ and $(R, \varphi, q) \notin C_2$. Then, by the form of D_2 it must be true that $w \in L(R^c) \Rightarrow w \notin L(R)$. Finally, as D_1 includes R we have that $(w, S) \not\models D_1$, which contradicts our initial supposition.

(iii)

$$D_1 \text{ is of the form } \bigcap_{R_i \in C_1} R_i \left(\bigwedge_{\varphi_i \in C_1} \varphi_i \wedge \bigwedge_{\varphi_i \notin C_1} \neg\varphi_i \right) \rightarrow \bigcup_{q \in C_1} \{q\}.$$
$$D_2 \text{ is of the form } \bigcap_{R_i \in C_2} R_i \cap \bigcap_{R_i \notin C_2} R_i^c \left(\bigwedge_{\varphi_i \in C_2} \varphi_i \right) \rightarrow \bigcup_{q \in C_2} \{q\}.$$

Here we have two more cases depending on the sets of rules of the initial automaton C_1 and C_2.

We abuse from the notation to state the fact that a pair (string, set of states) matches the left side of a rule.

In the case that $C_1 = C_2$ we have that it could be true that $(w, S) \models D_1$ and $(w, S) \models D_2$. However, the right side of the rule is the same for both rules and does not affect the the determinism of the automaton.

In the case that $C_1 \neq C_2$ we have two more cases.

(1) There is a pair (R, φ, q) such that $(R, \varphi, q) \in C_1$ and $(R, \varphi, q) \notin C_2$. Then, by the form of D_2 it must be true that $w \in L(R^c) \Rightarrow w \notin L(R)$. Finally, as D_1 includes R we have that $(w, S) \not\models D_1$, which contradicts our initial supposition.

(2) There is a pair (R, φ, q) such that $(R, \varphi, q) \in C_2$ and $(R, \varphi, q) \notin C_1$. Then, by the form of D_1 it must be true that $S \models \neg\varphi \Rightarrow S \not\models \varphi$. Finally, as D_2 includes φ we have that $(w, S) \not\models D_2$, which contradicts our initial supposition.

As every case leads to a contradiction we conclude that Δ^D must be deterministic. □

At this point we have our deterministic automaton, despite of this we still need to compute the emptiness algorithm over it. The way our procedure works is by computing all the reachable sets of states of the determinized transition relation and check if we reached a final state. The details of the procedure are shown in Algorithm 7. Besides, as we already proved that the determinization is correct, we provide a proof for correctness main loop of the algorithm (lines 11-40).

PROOF. By induction on the iterations, the pre and post conditions of the algorithm are defined bellow:

> **Preconditions.** The set of deterministic rules Δ^D and a list $Reachable\_States$, containing the sets of states reachable from the rules containing just a TRUE condition. Moreover, for every rule $\mathcal{A}(\varphi) \rightarrow S$, the automaton $\mathcal{A}$ is deterministic and the the formula φ is in disjunctive normal form.
>
> **Postconditions.** The list $Reachable\_States$, containing the sets of all the reachable states of the automaton.

Besides, let $I(k)$ be the invariant of the procedure defined as follows:

$I(k)$: At the end of k iterations every state that is in a set of $Reachable\_States$ is reachable.

We continue by induction on $I(k)$.

Base case. Before the loop starts, we have that $Reachable\_States$ contains all the states reachable from the rules starting with just a TRUE condition. As every state in the set is reachable $I(0)$ holds.

Inductive step. Let us assume that $I(k)$ holds and let $\mathcal{A}(\varphi) \to S$ be an arbitrary rule in Δ^D such that $L(\mathcal{A}) \neq \emptyset$. In line 13 the algorithm starts calculating the possible models for φ. In order to do this, the procedure iterates over the clauses of φ, let C be an arbitrary conjunctive clause.

$$C = \left(\bigwedge_{i=1}^{n} \exists q_i \right) \wedge \left(\bigwedge_{j=1}^{m} \forall q_j \right)$$

In between lines 15 and 22, the algorithm starts the construction of the minimal satisfying set for C. The construction of the minimal set is done as follows:

$$Minimal\_Set = \{A = \{q_i, q_1, \dots q_j, \dots, q_m\} | 1 \leq i \leq n\}$$

We abuse from the notation to state that each A_i is built by packing exactly one existential state and every single universal state. Regardless of this, by construction it is clear that $Minimal\_Set \models \varphi$. From line 23 to 24, the algorithm checks if the sets in $Reachable\_States$ suffice to build each A_i in $Minimal\_Set$. By simple inspection the procedure is correct, it simply checks if each A_i is a subset of some set in $Reachable\_States$. Then, if the set is buildable we add S to $Reachable\_States$. By inductive hypothesis, every state in $Reachable\_States$ must be reachable and, as we just used reachable states to construct $Minimal\_Set$, then S must be reachable and $I(k+1)$ holds.

Correctness. Let us suppose that $k = N$ and we are out of the loop. Then, by the condition of the loop it is not possible to add any more states to $Reachable\_States$. Then, as every reachable state is computed out by reachable states, and we exhausted all the possible states starting by the initial ones, the set must contain all of them and the postcondition must be true.

Finally, the algorithm simply checks if $\mathcal{F} \cap B \neq \emptyset$ for every $B \in Reachable\_Sets$. Finally, if a final state is reachable then the language must be nonempty. □

Last proof also give us insight about the complexity upper bound for the emptiness problem. Since the main loop ends when there are no changes in the set of reachable states, in the worst case

We can compute the path of reachable states giving as witness any string of the horizontal language.

the algorithm performs $|\Delta^D| \times |\Delta^D|$ iterations. Besides, as we know that the determinization gives our automaton a blow up of $2^{|\Delta|}$, we get our exponential blow up

$$O(|\Delta^D| \times |\Delta^D|) = O(2^{|\Delta|} \times 2^{|\Delta|}) = O(2^{2|\Delta|}) = O(2^{|\Delta|})$$

To conclude, as we know it is feasible to translate JSON Schema to QATA in polynomial time and then check the emptiness problem in EXPTIME, therefore the satisfiability problem for JSON Schema can be sorted out within the same upper bound.

□

Algorithm 7 Emptiness algorithm for QATA.

```
1:  procedure checkEmptiness(Q, Σ, F, Δ)
2:      Δ^D := Determinize(Δ)                          ▷ Determinize the rules of the automaton
3:      for (R, φ, S) ∈ Δ^D do
4:          A ← compute the DFA of R                   ▷ We need this to check emptiness of the horizontal language
5:          φ ← compute the DNF of φ                   ▷ We need this to build a minimal set for the formula
6:      end for
7:      Reachable_Sets = ∅                             ▷ Here we store the reachable sets
8:      for (A(TRUE) → S) ∈ Δ^D with L(A) ≠ ∅ do
9:          Reachable_Sets = Reachable_Sets ∪ {S}      ▷ We add the initial states
10:     end for
11:     repeat                                         ▷ Main loop, iterates exhaustively until no sets can be reached
12:         for (A(φ) → S) ∈ Δ^D with L(A) ≠ ∅ do
13:             for C ∈ Clauses(φ) do
14:                 Minimal_Set = ∅
15:                 for ∃q ∈ C do
16:                     A = ∅
17:                     A = A ∪ {q}
18:                     for ∀q ∈ C do
19:                         A = A ∪ {q}
20:                     end for
21:                     Minimal_Set = Minimal_Set ∪ {A}   ▷ We construct a set that satisfies the formula
22:                 end for
23:                 Buildable = TRUE
24:                 for A ∈ Minimal_Set do
25:                     Buildable_Aux = FALSE
26:                     for B ∈ Reachable_Sets do
27:                         if A ⊆ B then              ▷ Check if the sets can be constructed based on reachable states
28:                             Buildable_Aux = TRUE
29:                         end if
30:                     end for
31:                     if !Buildable_Aux then
32:                         Buildable = FALSE
33:                     end if
34:                 end for
35:                 if Buildable then
36:                     Reachable_Sets = Reachable_Sets ∪ {S}   ▷ If the set can be built, append it to the list
37:                 end if
38:             end for
39:         end for
40:     until No states can be added to Reachable_Sets
41:     for B ∈ Reachable_Sets do
42:         if F ∩ B ≠ ∅ then                          ▷ If a final state is reachable, we accept the automaton
43:             return TRUE
44:         end if
45:     end for
46:     return FALSE
```