Witness Generation for JSON Schema

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger

Witness Generation for JSON Schema

Lyes Attouche
Université Paris-Dauphine – PSL
lyes.attouche@dauphine.fr

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine – PSL
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dip. Informatica, Università di Pisa
ghelli@di.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

Stefanie Scherzinger
Universität Passau
stefanie.scherzinger@uni-passau.de

ABSTRACT

JSON Schema is a schema language for JSON documents, based on a complex combination of structural operators, Boolean operators (negation included), and recursive variables. The static analysis of JSON Schema documents comprises practically relevant problems, including schema satisfiability, inclusion, and equivalence. These problems can be reduced to witness generation: given a schema, generate an element of the schema — if it exists — and report failure otherwise. Schema satisfiability, inclusion, and equivalence have been shown to be decidable. However, no witness generation algorithm has yet been formally described. We contribute a first, direct algorithm for JSON Schema witness generation, and study its effectiveness and efficiency in experiments over several schema collections, including thousands of real-world schemas.

PVLDB Reference Format:
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Witness Generation for JSON Schema. PVLDB, 15(13): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.7106750.

1 INTRODUCTION

JSON Schema is a schema language based on a set of *assertions* that describe features of the JSON values described and on logical and structural combinators for these assertions.

While *validation* of a JSON value J with respect to a schema S, denoted $J \vDash S$, is a well-understood problem for which the JSON Schema Test Suite [25] lists over 50 validator tools at the time of writing, for the main static analysis problems, which we describe below, we still lack well-principled tools.

Inclusion $S \subseteq S'$: does, for each value J, $J \vDash S \Rightarrow J \vDash S'$? Checking schemas for inclusion (or containment) is of great practical importance: if the output format of a tool is specified by a schema S, and the input format of a different tool by a schema S', the problem of format compatibility is equivalent to schema inclusion $S \subseteq S'$; given the high expressive power of JSON Schema, this "format" may actually include detailed information about the range of specific parameters. For example, the IBM ML framework LALE [15] adopts an incomplete inclusion checking algorithm for JSON Schema, to improve safety of ML pipelines [22].

Equivalence $S \equiv S'$: does, for each value J, $J \vDash S \Leftrightarrow J \vDash S'$? Checking equivalence builds upon inclusion, and is relevant in designing workbenches for schema analysis and simplification [21].

Satisfiability of S: does a value J exist such that $J \vDash S$?

Witness generation for S, a constructive generalization of satisfiability: given S, generate a value J such that $J \vDash S$, or return "unsatisfiable" if no such value exists. In the first case, we call J a *witness*. Schema inclusion $S \subseteq S'$ can be immediately reduced to witness generation for $S \wedge \neg S'$, but with a crucial advantage: if a witness J for $S \wedge \neg S'$ is generated, we can provide users with an explanation: S is not included in S' *because* of values such as J. We can similarly solve a "witnessed" version of equivalence: given S and S', either prove that one is equivalent to the other, or provide an explicit witness J that belongs to one, but not to the other.

The techniques and the notions that we present in this paper can be also useful for the design of *example generation* algorithms, that is, algorithms that do not just generate one arbitrary witness, but generate many of them, according to some heuristics aimed to fulfill criteria of "completeness" and "realism".

Open challenges. Witness generation for JSON Schema is difficult. Existing tools are incomplete and struggle with this task (as we will show in our experiments). First of all, JSON Schema includes conjunction, disjunction, negation, modal (or *structural*) operators, recursive second-order variables, and recursion under negation. Secondly, for each JSON type, the different structural operators have complex interactions, as in the following example, where `"required"` and the negated `"patternProperties"` force the presence of fields whose names match `"^a"` and `"^abz$"` (this is explained in the paper), `"maxProperties" : 1` forces these two fields to be one, and, finally, `"patternProperties"` forces the value of that field to satisfy *var2*, since `"abz"` also matches `"z$"`.

```
{"required":["abz"],
 "not":{"patternProperties":{"^a":{"$ref":"#/$defs/var1"}}},
 "maxProperties":1,
 "patternProperties":{"z$":{"$ref":"#/$defs/var2"}},
 "$defs" : ...
}
```

Each aspect would make the problem computationally intractable by itself. Their combination exacerbates the difficulty of the

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:XX.XX/XXX.XX

design of a *complete* algorithm that is *practical*, that is, of an algorithm that is correct and complete by design, but is also able to run in a reasonable time over the vast majority of real-world schemas.

Contributions. The main contribution of this paper is an original sound and complete algorithm for checking the satisfiability of an input schema S, generating a witness J when the schema is satisfiable. Our algorithm supports the whole language without uniqueItems. While the existence of an algorithm for this specific problem follows from the results in [17], where the problem is proved to be EXPTIME-complete, we are the first to explicitly describe an algorithm, and specifically one that has the potential to work in reasonable time over schemas of realistic size. Our algorithm is based on a set of formal manipulations of the schema, some of which, such as *preparation*, are unique to JSON Schema, and have not been proposed before in this form. Particularly relevant in this context is the notion of *lazy and-completion*, which we will describe later. In this paper, we detail each algorithm phase, show that each is in $O(2^{poly(N)})$, and focus on preparation and generation of objects, the phases completely original to this work.

The practical applicability of our algorithm is proved by our experimentation, which is another contribution of this work. Our experiments are based on four real-world datasets, on a synthetic dataset, and on a handwritten dataset. Real-world datasets comprise 6,427 unique schemas extracted, through an extensive data cleaning process, from a large corpus of schemas crawled from GitHub [13] and curated by us for errors and redundancies; the other datasets, already used in [22], are related to specific application domains and originated from Snowplow[4], The Washington Post [27], and Kubernetes [23]. The synthetic dataset is synthesized from the standard schemas provided by *JSON Schema Org* [25], from which we derive schemas that are known to be satisfiable or unsatisfiable by design [6]. The handwritten dataset is specifically engineered to test the most complex aspects of the JSON Schema language. The experiments show that our algorithm is complete, and that, despite its exponential complexity, it behaves quite well even on schemas with tens of thousands of nodes.

Paper outline. In Section 2 we review related work. In Section 3 we describe our algebraic representation of JSON Schema. In Sections 4–6, we describe the algorithm. In Section 7 we present our experimental evaluation. We conclude in Section 8.

2 RELATED WORK

Overviews over schema languages for JSON can be found in [10, 11, 17, 26]. Pezoa et al. [26] introduced the first formalization of JSON Schema and showed that it cannot be captured by MSO or tree automata because of the uniqueItems constraints. While they focused on validation and proved that it can be decided in $O(|J|^2|S|)$ time, they also showed that JSON Schema can simulate tree automata. Hence, schema satisfiability is EXPTIME-hard.

In [17] Bourhis et al. refined the analysis of Pezoa et al. They mapped JSON Schema onto an equivalent modal logic, called recursive JSL, and proved that satisfiability is EXPTIME-complete for recursive schemas without uniqueItems, and it is in 2EXPTIME for recursive schemas with uniqueItems. Their work is extremely important in establishing complexity bounds. Since they map JSON Schema onto recursive JSL logic, and provide a specific kind of alternating tree automata for this logic, they already provide an indirect indication of an algorithm for witness generation. However, classical reachability algorithms for alternating automata are designed to prove complexity upper bounds, not as practical tools. They are typically based on the exploration of all subsets of the state set of the automaton [19], hence on a sequence of complex operations on a set of sets whose dimension may be in the realm of $2^{10,000}$. While exponentiality cannot be avoided in the worst case, it is clear that we need a different approach when designing a practical algorithm.

To the best of our knowledge, the only tool that is currently available to check the satisfiability of a schema is the containment checker described by Habib et al. [22]. While it has been designed for schema containment checking, e.g., $S_1 \subseteq S_2$, it can also be exploited for schema satisfiability since S is satisfiable if and only if $S \not\subseteq S'$, where S' is an empty schema. The approach of Habib et al. bears some resemblances to ours, e.g., schema canonicalization has been first presented there, but its ability to cope with negation is very limited as well as its support for recursion.

Several tools (see [16] and [1]) for example generation exist. They generate JSON data starting from a schema. These tools, however, are based on a trial-and-error approach and cannot detect unsatisfiable schemas. We compare our tool with [16] in our experiments.

Own prior work. In our technical report [14], we discuss negation-completeness for JSON Schema, that is, we show how pairs of schema operators such as "patternProperties"-"required" and "items"-"contains" are *almost* dual under negation, as $\wedge$-$\vee$ or $\forall$-$\exists$ are, but not exactly. In the process, we define an algorithm for not-elimination.

A preliminary version of the algorithm described in the current paper has been presented in [12]. In that paper we provided an hint on the different phases of the algorithm, while here we go in much more detail. An earlier prototype implementation has been presented in tool demos [7, 8, 21].

This paper is accompanied by a full version [9], containing detailed proofs and additional experiments.

3 JSON SCHEMA AND THE ALGEBRA

3.1 JSON data model

Each JSON value belongs to one of the six JSON Schema types: nulls, Booleans, decimal numbers Num, strings Str, objects, arrays.

J ::=	$B \mid O \mid A$		**JSON expr**
B ::=	null $\mid$ true $\mid$ false $\mid q \mid s$		
		$q \in \mathsf{Num}, s \in \mathsf{Str}$	**Basic values**
O ::=	$\{l_1 : J_1, \ldots, l_n : J_n\}$		
		$n \geq 0, i \neq j \Rightarrow l_i \neq l_j$	**Objects**
A ::=	$[J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

Definition 1 (JSON objects). We interpret a JSON object $\{l_1 : J_1, \ldots, l_n : J_n\}$ as a *set* of pairs (*members*) $\{(l_1, J_1), \ldots, (l_n, J_n)\}$, where $i \neq j \Rightarrow l_i \neq l_j$, and an array $[J_1, \ldots, J_n]$ as an ordered list; JSON value equality is defined accordingly, that is, by ignoring member order when comparing objects.

3.2 JSON Schema

We base our work on JSON Schema Draft-06 [31], as it supports virtually all schemas that we could crawl from GitHub [13]. The successive Draft 2019-09 [30] made validation dependent on annotations, a questionable semantic shift that we prefer not to embrace for now. However, we include in our algebra the operators `"minContains"` and `"maxContains"` introduced with Draft 2019-09, since they are very interesting in the context of witness generation, and their semantics does not depend on annotations.

JSON Schema uses JSON syntax. A schema is a JSON object that collects *assertions* that are members, i.e., name-value pairs, where the name indicates the assertion and the value collects its parameters, as in `"minLength" : 3`, where the value is a number, or in `"items" : {"type" : ["boolean"]}`, where the value for `"items"` is an object that is itself a schema. We next describe JSON Schema by giving its translation into an algebra.

3.3 The *core* and the *positive* algebras

In JSON Schema, the meaning of some assertions is modified by the surrounding assertions, making formal manipulation much more difficult. Moreover, the language is rich in redundant operators. In our implementation, we therefore map an input schema onto an algebraic representation based on a *core algebra*, an algebraic version of JSON Schema with less redundant operators. We then eliminate negative expressions through not-elimination (Section 5.2), by using a *positive algebra* without negation but with three new operators: $\mathsf{notMulOf}(n)$, $\mathsf{pattReq}(r : S)$, and $\mathsf{contAfter}(i^+ : S)$.

Our algebras extend JSON Schema regular expressions with external intersection and complement operators $r \sqcap r'$ and $\overline{r}$; this extension is discussed in Section 3.4. The syntax of the two algebras, *core* and *positive*, is presented below.

$$
\begin{array}{lll}
\multicolumn{3}{l}{m \in \mathsf{Num}^{-\infty}, M \in \mathsf{Num}^{\infty}, l \in \mathbb{N}_{>0}, i \in \mathbb{N}, j \in \mathbb{N}^{\infty}, q \in \mathsf{Num}, k \in \mathsf{Str}} \\
T & ::= & \mathsf{Arr} \mid \mathsf{Obj} \mid \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Num} \\
r & ::= & \text{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \\
S & ::= & \mathsf{ifBoolThen}(b) \mid \mathsf{pattern}(r) \mid \mathsf{betw}_m^M \mid \mathsf{xBetw}_m^M \\
 & & \mid \mathsf{mulOf}(q) \mid \mathsf{props}(r : S) \mid \mathsf{req}(k) \mid \mathsf{pro}_i^j \\
 & & \mid \mathsf{item}(l : S) \mid \mathsf{items}(i^+ : S) \mid \mathsf{cont}_i^j(S) \\
 & & \mid \mathsf{type}(T) \mid x \mid\ S_1 \wedge S_2 \mid S_1 \vee S_2 \\
\text{core:} & & \mid \neg S \\
\text{positive:} & & \mid \mathsf{notMulOf}(q) \mid \mathsf{pattReq}(r : S) \mid \mathsf{contAfter}(i^+ : S) \\
E & ::= & x_1 : S_1, \ldots, x_n : S_n \\
D & ::= & S \text{ defs } (E)
\end{array}
$$

$\mathsf{Num}^{-\infty}$ are the decimal numbers extended with $-\infty$, and similarly for Num^{∞} and $\mathbb{N}^{\infty}$. $\mathbb{N}_{>0}$ is $\mathbb{N}$ without 0, used in $\mathsf{item}(l : S)$.

We distinguish Boolean operators ($\wedge$, $\vee$ and $\neg$), variables (x), and *Typed Operators* (TO — all the others). All TOs different from $\mathsf{type}(T)$ have an implicative semantics: "if the instance belongs to the type T then …", so that they are trivially satisfied by every instance not belonging to type T. We say that they are *implicative typed operators* (ITOs).

The operators of the core algebra strictly correspond to those of JSON Schema, and in particular to their implicative semantics.

Informally, an instance J of the core or positive algebra satisfies an assertion S if:

- $\mathsf{ifBoolThen}(b)$: *if* the instance J is a boolean, *then* $J = b$.
- $\mathsf{pattern}(r)$: *if* J is a string, *then* J matches r.
- betw_m^M: *if* J is a number, *then* $m \le J \le M$. xBetw_m^M is the same with extreme excluded.
- $\mathsf{mulOf}(q)$: *if* J is a number, *then* $J = q \times i$ for some integer i. q is any number, i.e., any *decimal* number (Section 3.1).
- $\mathsf{props}(r : S)$ *if* J is an object and *if* (k, J') is a member of J where k matches the pattern r, *then* J' satisfies S. Hence, it is satisfied by any instance that is not an object and also by any object where no member name matches r.
- $\mathsf{req}(k)$: *if* J is an object, *then* it contains at least one member whose name is k.
- pro_i^j: *if* J is an object, *then* it has between i and j members.
- $\mathsf{item}(l : S)$: *if* J is an array $[J_1, \ldots, J_n]$ $(n \ge 0)$ and *if* $l \le n$, *then* J_l satisfies S. Hence, it is satisfied by any J that is not an array and also by any array that is strictly shorter than l: it does not force the position l to be actually used.
- $\mathsf{items}(i^+ : S)$: *if* J is an array $[J_1, \ldots, J_n]$, *then* J_l satisfies S for every $l > i$. Hence, it is satisfied by any J that is not an array and by any array shorter than i.
- $\mathsf{cont}_i^j(S)$: *if* J is an array, *then* the total number of elements that satisfy S is included between i and j.
- $\mathsf{type}(T)$ is satisfied by any instance belonging to the predefined JSON type T (Str, Num, Bool, Obj, Arr, and Null).
- x is equivalent to its definition in the environment E associated with the expression.
- $S_1 \wedge S_2$: both S_1 and S_2 are satisfied.
- $S_1 \vee S_2$: either S_1, or S_2, or both, are satisfied.
- $\neg S$: S is not satisfied.
- $\mathsf{notMulOf}(n)$: *if* J is a number, *then* is not a multiple of n.
- $\mathsf{pattReq}(r : S)$: *if* J is an object, *then* it contains at least one member (k, J) where k matches r and J satisfies S
- $\mathsf{contAfter}(i^+ : S)$: *if* J is an array $[J_1, \ldots, J_n]$, *then* it contains at least one element J_j with $j > i$ that satisfies S.
- $D = S$ defs $(x_1 : S_1, \ldots, x_n : S_n)$: J satisfies S when every x_i is interpreted as an alias for the corresponding S_i.

Variables in $E = x_1 : S_1, \ldots, x_n : S_n$ are mutually recursive, but we require recursion to be *guarded*. Let us say that x_i *directly depends* on x_j if some occurrence of x_j appears in the definition of x_i without being in the scope of an ITO. Recursion is not guarded if the transitive closure of the relation "directly depends on" contains a reflexive pair (x, x). Informally, recursion is guarded iff every cyclic chain of dependencies traverses an ITO. An environment $E = x_1 : S_1, \ldots, x_n : S_n$ is *guarded* if recursion is guarded in E. An environment $E = x_1 : S_1, \ldots, x_n : S_n$ is *closing* for S if all variables in $S_1, \ldots, S_n$ and in S are included in $x_1, \ldots, x_n$.

The three operators added in the positive algebra do not directly correspond to JSON Schema operators, but can still be expressed in JSON Schema, through the negation of mulOf, props, and items, as follows, where $S_1 \Rightarrow S_2$ is an abbreviation for $\neg S_1 \vee S_2$:

$$
\begin{array}{lcl}
\mathsf{notMulOf}(n) & = & \mathsf{type}(\mathsf{Num}) \Rightarrow \neg\mathsf{mulOf}(n) \\
\mathsf{pattReq}(r : S) & = & \mathsf{type}(\mathsf{Obj}) \Rightarrow \neg\mathsf{props}(r : \neg S) \\
\mathsf{contAfter}(i^+ : S) & = & \mathsf{type}(\mathsf{Arr}) \Rightarrow \neg\mathsf{items}(i^+ : \neg S)
\end{array}
$$

In [9] we formalize the official JSON Schema semantics by defining a function $[[S]]_E$ that associates a set of JSON values to any assertion S whose variables are defined by the guarded schema E, also in cases of mutual recursion under negation.

Hereafter we will often use the redundant operators **t** and **f**, where **t** is satisfied by any JSON value, and **f** is satisfied by none.

3.4 About regular expressions

3.4.1 Mapping JSON Schema regular expressions onto standard REs. Following the example of [17], we represent JSON Schema regular expressions (REs) using standard REs. In practice, in our implementation we map every JSON Schema RE into a standard RE, using a simple incomplete algorithm,[1] and we are currently able to translate more than 97% of the distinct patterns in our corpus. The others mostly contain look-ahead and look-behind.

3.4.2 Extending REs with external complement and intersection. In our algebra, we use a form of *externally extended REs* (EEREs), where the two extra operators are not first class RE operators, so that one cannot write $(\overline{r})*$, but they can be used at the outer level:

$$r ::= \textit{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2$$

This extension does not affect the expressive power of regular expressions but affects their succinctness, hence the complexity of problems such as emptiness checking. We are going to exploit this expressive power in four different ways:

(1) in order to translate "additionalProperties" : S as $\text{props}(\overline{(r_1| \ldots |r_m)} : \langle S \rangle)$ (Section 3.5);
(2) in order to translate "propertyNames" : S (Section 3.5);
(3) during not-elimination (Section 5.2), where $\text{pattern}(\overline{r})$ is used to rewrite $\neg\text{pattern}(r)$;
(4) during object preparation (Section 6.3.3), where we must express the intersection and the difference of patterns that appear in $\text{props}(r : S)$ and $\text{pattReq}(r : S)$ operators.

During the final phases of our algorithm (Section 6.3), we need to solve the following *i-enumeration* problem (which generalizes emptiness) for our EEREs: for a given EERE r and for a given i, either return i words that belong to $L(r)$, where $L(r)$ is the language of r, or return "impossible" if $|L(r)| < i$. It is well-known that emptiness of REs extended (internally) with negation and intersection is non-elementary [28]. However, in the full paper [9] we show that for our external-only extension i-enumeration and emptiness can be solved in time $O(i^2 \times 2^n)$.

3.5 From JSON Schema to the Algebra

The translation from JSON Schema to the algebra is rather intuitive, and is described in [9]. Essentially, each JSON Schema assertion is translated into the corresponding algebraic assertion. However, attention must be paid to certain families of assertions, which must be grouped and translated together:

- `if`, `then`, `else`, translated using Boolean operators;
- property assertions `additionalProperties`, `properties`, and `patternProperties` (here indicated as `adPr`, `pr`, and `paPr`): `pr`, `paPr` correspond to our props operator, while "adPr" : S associates a schema S to any name that does not match either `pr` or `paPr` arguments, and is translated as $\text{props}(\overline{(r_1| \ldots |r_m)} : S)$, where $r_1, \ldots, r_m$ are patterns that represent all arguments of all `pr` or `paPr` that occur in the same schema;
- `additionalItems`, `items`, translated using the algebra assertion $\text{items}(j^+ : S)$, $\text{item}(l : S)$;
- `minContains`, `maxContains`, `contains`, which are translated as $\text{cont}_m^M(\ldots)$.

Some redundant operators are mapped to simpler operators:

- "oneOf" : $[S_1, \ldots, S_n]$ requires that a value J satisfies one of $S_1, \ldots, S_n$ and violates all the others; it is translated using Boolean operators and variables;
- "propertyNames" : S requires that every member name satisfies S; it is translated as $\text{props}(\textit{PattOfS}(\neg S) : \mathbf{f})$, where $\textit{PattOfS}(\neg S)$ is a pattern that uses $\overline{r}$ and $\sqcap$ in order to encode all strings that violate S;
- the "dependencies" assertion specifies that if the instance contains a member with name k_i, then it must also satisfy some other assertions; it is translated using req and $\Rightarrow$;
- "const" : J and "enum" : $[J_1, \ldots, J_n]$, used to restrict a schema to a finite set of values; they are translated to structural operators as in [22].

Finally, the definitions-references mechanism of JSON Schema (the $ref : *path* operator) is translated into our simpler mechanism, based on variables and environments.

[1] Dominik Freydenberger suggested this algorithm to us, in personal communication.

4 THE STRUCTURE OF THE ALGORITHM

In a recursive algorithm for witness generation, in order to generate a witness for an ITO such as $\text{pattReq}(r : S)$, one can generate a witness J for S and use it to build an object with a member whose name matches r and whose value is J. The same approach can be followed for the other ITOs. For the Boolean operator $S_1 \vee S_2$, one recursively generates witnesses of S_1 and S_2.

Negation and conjunction are much less direct: there is no way to generate a witness for $\neg S$ starting from a witness for S. Also, given a witness for S_1, if it is not a witness for $S_1 \wedge S_2$, we may need to try infinitely many others before finding one that satisfies S_2 as well. We solve this problem as follows. We first eliminate $\neg$ using not-elimination, then we bring all definitions of variables into DNF so that conjunctions are limited to sets of ITOs that regard the same type (Section 5). We then perform a form of *and-elimination* over these homogeneous conjunctions (*preparation*), and we finally use these "prepared" homogeneous conjunctions to generate the witnesses, through a bottom-up iterative process (Section 6).

Preparation is the crucial step: here we make all the interactions between the conjuncted ITOs explicit, which may require the generation of new variables. This phase is delicate because it is exponentially hard in the general case, and we must organize it in order to run fast enough in typical case. Moreover, it may generate infinitely many new variables, which we avoid with a technique based on ROBDDs, that we define in Section 5.1.

5 TRANSFORMATION IN POSITIVE, STRATIFIED, GROUND, CANONICAL DNF

We will illustrate the preliminary phases of our algorithm by exploiting the running example of Figure 1.

(a)
$r : \text{pattReq}(b : x) \vee \text{props}(a : y) \vee \text{props}(a.* : \neg r \vee x),$
$x : \text{type}(\text{Arr}), \qquad y : \text{type}(\text{Num})$

(b)
$r : \text{pattReq}(b : x) \vee \text{props}(a : y) \vee \text{props}(a.* : co(r) \vee x),$
$x : \text{type}(\text{Arr}), \qquad y : \text{type}(\text{Num}),$
$co(r) : \text{type}(\text{Obj}) \wedge \text{props}(b : co(x)) \wedge \text{pattReq}(a : co(y))$
$\wedge \text{pattReq}(a.* : r \wedge co(x)),$
$co(x) : \text{type}(\text{Null}) \vee \text{type}(\text{Bool}) \vee \text{type}(\text{Num}) \vee \text{type}(\text{Str}) \vee \text{type}(\text{Obj}),$
$co(y) : \text{type}(\text{Null}) \vee \text{type}(\text{Bool}) \vee \text{type}(\text{Str}) \vee \text{type}(\text{Obj}) \vee \text{type}(\text{Arr})$

(c)
$r : \text{pattReq}(b : x) \vee \text{props}(a : y) \vee \text{props}(a.* : crx),$
$co(r) : \text{type}(\text{Obj}) \wedge \text{props}(b : co(x)) \wedge \text{pattReq}(a : co(y))$
$\wedge \text{pattReq}(a.* : rcx),$
$crx : co(r) \vee x, \qquad rcx : r \wedge co(x)$

(d)
$crx : \{\text{type}(\text{Obj}), \text{props}(b : co(x)), \text{pattReq}(a : co(y)),$
$\text{pattReq}(a.* : rcx)\} \quad \vee \quad \{\text{type}(\text{Arr})\},$
$rcx : \{(\text{pattReq}(b : x), \text{type}(\text{Null})\} \vee \{(\text{pattReq}(b : x), \text{type}(\text{Bool})\}$
$\vee \{(\text{pattReq}(b : x), \text{type}(\text{Num})\} \vee \{(\text{pattReq}(b : x), \text{type}(\text{Str})\}$
$\vee \{(\text{pattReq}(b : x), \text{type}(\text{Obj})\}$
$\vee \{\text{props}(a : y), \text{type}(\text{Null})\} \vee \{\text{props}(a : y), \text{type}(\text{Bool})\}$
$\vee \{\text{props}(a : y), \text{type}(\text{Num})\} \vee \{\text{props}(a : y), \text{type}(\text{Str})\}$
$\vee \{\text{props}(a : y), \text{type}(\text{Obj})\}$
$\vee \{\text{props}(a.* : \text{crx}), \text{type}(\text{Null})\} \vee \ldots$
$\vee \{\text{props}(a.* : \text{crx}), \text{type}(\text{Obj})\}$

(e)
$r : \{\text{type}(\text{Obj}), \text{pattReq}(b : x)\} \vee \{\text{type}(\text{Obj}), \text{props}(a : y)\}$
$\vee \{\text{type}(\text{Obj}), \text{props}(a.* : crx\} \vee \{\text{type}(\text{Null})\}$
$\vee \{\text{type}(\text{Bool})\} \vee \{\text{type}(\text{Num})\} \vee \{\text{type}(\text{Str})\} \vee \{\text{type}(\text{Arr})\},$
$rcx : \{\text{type}(\text{Obj}), \text{pattReq}(b : x)\} \vee \{\text{type}(\text{Obj}), \text{props}(a : y)\}$
$\vee \{\text{type}(\text{Obj}), \text{props}(a.* : crx)\} \vee \{\text{type}(\text{Null})\}$
$\vee \{\text{type}(\text{Bool})\} \vee \{\text{type}(\text{Num})\} \vee \{\text{type}(\text{Str})\} \vee \{\text{type}(\text{Arr})\}$

Figure 1: (a) Original term. (b) After not-elimination. (c) After stratification, omitting unaffected variables. (d) After transformation to GDNF. (e) After canonicalization.

5.1 Premise: ROBDD reduction

Two expressions built with variables and Boolean operators are *Boolean-equivalent* when they can be proved equivalent using the laws of the Boolean algebra. A Reduced Ordered Boolean Decision Diagram (ROBDD) is a data structure that provides the same representation for two such expressions if, and only if, they are Boolean-equivalent [18]. Hence, whenever we define a variable x whose body S_x is a Boolean combination of variables, in any phase of the algorithm, we perform the *ROBDD reduction*: we compute the ROBDD representation of S_x, $robdd(S_x)$, and we store a pair $x : robdd(S_x)$ in the ROBDDTab table, unless a pair $y : robdd(S_y)$ with $robdd(S_x) = robdd(S_y)$ is already present. In this case, we substitute every occurrence of x with y. This technique makes the entire algorithm more efficient and, crucially, it ensures termination of the preparation phase (Section 6.3.3).

5.2 Not-elimination

Not-elimination, described in detail in [14], proceeds in two phases: *not-completion* of variables and *not-rewriting*.

During not-completion of variables, for every variable $x_n : S_n$ we define a corresponding $not\_x_n : \neg S_n$. After not-completion, every variable has a complement variable $co(x_i) = not\_x_i$ and $co(not\_x_i) = x_i$. The complement $co(x)$ is used for not-elimination.

During not-rewriting, we rewrite every expression $\neg S$ into an expression where the negation has been pushed inside, by applying the rules reported in [14], which include the following (Fig. 1(b)):

$$
\begin{array}{lcl}
\neg(\text{pattern}(r)) & = & \text{type}(\text{Str}) \wedge \text{pattern}(\overline{r}) \\
\neg(\text{props}(r : S)) & = & \text{type}(\text{Obj}) \wedge \text{pattReq}(r : \neg S) \\
\neg(\text{pattReq}(r : S)) & = & \text{type}(\text{Obj}) \wedge \text{props}(r : \neg S) \\
\neg(\text{item}(l : S)) & = & \text{type}(\text{Arr}) \wedge \text{item}(l : \neg S_i) \wedge \text{cont}_l^{\infty}(\text{t}) \\
\neg(\text{items}(i^+ : S)) & = & \text{type}(\text{Arr}) \wedge \text{contAfter}(i^+ : \neg S) \\
\neg(\text{contAfter}(i^+ : S)) & = & \text{type}(\text{Arr}) \wedge \text{items}(i^+ : \neg S) \\
\neg(x) & = & co(x)
\end{array}
$$

Not-elimination of a schema of size N produces an equivalent schema of size $O(N)$ [14]; hereafter, we use N to indicate the size of the abstract syntax tree of the original schema, where numbers and strings are represented in binary notation, assuming that the i and j constants different from ∞ that appear in $\text{item}(i : S)$, $\text{items}(i^+ : S)$, $\text{contAfter}(i^+ : S)$, $\text{cont}_i^j(S)$, and pro_i^j, are smaller than the input size. This assumption is justified by the observation that in practical cases these numbers tend to be extremely small in comparison to the input size.

5.3 Stratification and Transformation in Canonical Guarded DNF

Stratification. We say that a schema is *stratified* when every schema argument of every ITO is a variable, so that $\text{pattReq}(a : x \wedge y)$ is not stratified while $\text{pattReq}(a : w)$ is stratified.

During stratification, for every ITO that has a subschema S in its syntax, such as $\text{cont}_i^j(S)$, when S is not a variable, we create a new variable $x : S$, we substitute S with x, and we apply not-completion, not-elimination and ROBDD reduction to $x : S$ (Figure 1(c)).

Stratification of a schema of size N produces an equivalent schema of size $O(N)$ [9].

Guarded Disjunctive Normal Form (GDNF). A schema is in GDNF if it has the shape $\vee(\wedge(S_{1,1}, \ldots, S_{1,n_1}), \ldots, \wedge(S_{l,1}, \ldots, S_{l,n_l}))$ and every $S_{i,j}$ is a TO. To transform an environment E into a corresponding one E^G in GDNF, we just substitute every variable that is not in the scope of an ITO with its body, a process that is guaranteed to terminate, thanks to the guardedness condition on E, and we bring the result in DNF by distributivity of $\wedge$ and $\vee$ (Figure 1(d)); hereafter, for brevity, we use $\{S_1, \ldots, S_n\}$ to indicate a conjunction $S_1 \wedge \ldots \wedge S_n$. Reduction to GDNF can lead to an exponential explosion, and is actually a very expensive phase, according to our experiments (Section 7): observe, in Figure 1(d), how the size of rcx is the product of the sizes of r and that of $co(x)$.

Canonicalization. We say that a conjunction that contains exactly one assertion $\text{type}(T)$ and a set of ITOs of that same type T is a *typed group* of type T; canonicalization splits every conjunct of the GDNF into a set of *typed groups* (Figure 1(e), where we also applied elementary equivalences, such as idempotence of $\vee$).

By construction, every phase described in this section transforms a JSON Schema document into an equivalent one.

Property 1 (Equivalence). *The phases of not-elimination, stratification, transformation into Canonical GDNF, transform a JSON Schema document into an equivalent one.*

$$
\begin{array}{lcl}
\langle\!\langle x \rangle\!\rangle_A & = & A(x) \\
\langle\!\langle \mathsf{ifBoolThen}(b) \rangle\!\rangle_A & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Bool}) \Rightarrow J = b \,|\!\} \\
\langle\!\langle \mathsf{props}(r : S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \\
 & & \forall i \in \{1..n\}.\ k_i \in L(r) \ \Rightarrow J_i \in \langle\!\langle S \rangle\!\rangle_A \,|\!\} \\
\langle\!\langle \mathsf{item}(l : S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & n \geq l \Rightarrow J_l \in \langle\!\langle S \rangle\!\rangle_A \,|\!\} \\
\langle\!\langle \mathsf{items}(i^+ : S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & \forall j \in \{1..n\}.\ j > i \Rightarrow J_j \in \langle\!\langle S \rangle\!\rangle_A \,|\!\} \\
\langle\!\langle \mathsf{cont}_i^j(S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & i \leq \ |\{\!|\, l \mid J_l \in \langle\!\langle S \rangle\!\rangle_A \,|\!\}| \ \leq j \,|\!\} \\
\langle\!\langle S_1 \wedge S_2 \rangle\!\rangle_A & = & \langle\!\langle S_1 \rangle\!\rangle_A \cap \langle\!\langle S_2 \rangle\!\rangle_A \\
\langle\!\langle S_1 \vee S_2 \rangle\!\rangle_A & = & \langle\!\langle S_1 \rangle\!\rangle_A \cup \langle\!\langle S_2 \rangle\!\rangle_A \\
\langle\!\langle \mathsf{pattReq}(r : S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = \{\!|(k_1 : J_1), \ldots, (k_n : J_n)|\!\} \Rightarrow \\
 & & \exists i \in \{1..n\}.\ k_i \in L(r)) \ \wedge J_i \in \langle\!\langle S \rangle\!\rangle_A \,|\!\} \\
\langle\!\langle \mathsf{contAfter}(i^+ : S) \rangle\!\rangle_A & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & \exists l.\ l > i \ \wedge J_l \in \langle\!\langle S \rangle\!\rangle_A \,|\!\}
\end{array}
$$

Figure 2: Rules for assignment evaluation.

6 PREPARATION AND GENERATION

6.1 Assignments and bottom-up semantics

We define an assignment A for an environment E as a function mapping each variable of E to a set of JSON values. An assignment is sound when it maps each variable y to a subset of its semantics $[\![y]\!]_E$, which denotes the set of JSON values that satisfy y in E.

Definition 2 (Assignments, Soundness). An assignment A for an environment E is a function mapping each variable of E to a set of JSON values. An assignment A for E is sound iff for all $y \in Vars(E)$: $A(y) \subseteq [\![y]\!]_E$.

Given a schema S defs (E) and an assignment A for E, we can evaluate S using A to interpret any variable in S, by applying the rules exemplified in Figure 2 (see [9] for a complete list), where universal quantification on an empty set is true, the set $\{1..0\}$ is empty, and $\mathcal{J}Val(\mathsf{Bool})$ is the set of JSON values of type Bool. For example, if $A(x) = \{\!|J|\!\}$, and if $S = \{\mathsf{type}(\mathsf{Arr}), \mathsf{items}(0^+ : x), \mathsf{cont}_1^2(\mathbf{t})\}$, then $\langle\!\langle S \rangle\!\rangle_A = \{\!|[J], [J, J]|\!\}$. Intuitively, $\langle\!\langle S \rangle\!\rangle_A$ uses the witnesses collected by A in order to build bigger witnesses for S.

Hence, the repeated application of these rules, starting from an empty assignment $\mathcal{A}_E^0$, defines a sequence of assignments $\mathcal{A}_E^i$ containing more and more witnesses, whose limit $\mathcal{A}_E^\infty$ defines a bottom-up semantics for JSON Schema.

Definition 3 ($\mathcal{A}_E^i$, $\mathcal{A}_E^\infty$). For a given positive environment E, the sequence of assignments $\mathcal{A}_E^i$ is defined as:

$$
\begin{array}{ll}
\forall y \in Vars(E): & \mathcal{A}_E^0(y) = \emptyset \\
\forall y \in Vars(E): & \mathcal{A}_E^{i+1}(y) = \langle\!\langle E(y) \rangle\!\rangle_{\mathcal{A}_E^i}
\end{array}
$$

The assignment $\mathcal{A}_E^\infty$ is defined as $\bigcup_{i \in \mathbb{N}} \mathcal{A}_E^i$.

The function $[\![S]\!]_E$ described in the full paper [9] corresponds to the official semantics, and is based on the top-down substitution of variables with their definitions during the validation of a JSON value. In [9], we show that, on positive schemas, the bottom-up interpretation $\langle\!\langle S \rangle\!\rangle_{\mathcal{A}_E^\infty}$ corresponds to $[\![S]\!]_E$.

Any JSON value J has a *depth* $\delta(J)$, that is the number of levels of its tree representation, formally defined as follows.

Definition 4 (Depth $\delta(J)$, $\mathcal{J}^d$). The depth of a JSON value J, $\delta(J)$, is defined as follows, where $\max(\{\!|\,|\!\})$ is defined to be 0:

$$
\begin{array}{ll}
J \text{ belongs to a base type}: & \delta(J) = 1 \\
J = [J_1, \ldots, J_n]: & \delta(J) = 1 + \max(\{\!|\delta(J_1), \ldots, \delta(J_n)|\!\}) \\
J = \{a_1 : J_1, \ldots, a_n : J_n\}: & \delta(J) = 1 + \max(\{\!|\delta(J_1), \ldots, \delta(J_n)|\!\})
\end{array}
$$

$\mathcal{J}^d$ is the set of all JSON values J with $\delta(J) \leq d$.

The assignment $\mathcal{A}_E^i$ includes all witnesses of depth i: for any depth i, it can be proved that $([\![y]\!]_E \cap \mathcal{J}^i) \subseteq \mathcal{A}_E^i(y)$.

6.2 Bottom-up iterative witness generation

Since S defs (E) is equivalent to x defs $(x : S, E)$, we will discuss here, for simplicity, generation for the x defs (E) case.

Our algorithm for bottom-up iterative witness generation for a schema x defs (E) produces a sequence of finite assignments A^i, each approximating the assignment $\mathcal{A}_E^i$, until we reach either a witness for x or an "unsatisfiability fix-point", which is a notion that we will introduce shortly.

A^i is built as follows: $A^0 = \mathcal{A}_E^0$; then, at step i, for each $y \in Vars(E)$, we compute a set of new values for y based on the current assignment A^i by using a generation algorithm $\mathsf{Gen}(E(y), A^i)$ that computes a subset of $\langle\!\langle E(y) \rangle\!\rangle_{A^i}$; formally, $A^{i+1}(y) = \mathsf{Gen}(E(y), A^i)$. Our specific Gen algorithm is defined in the next section, but we show now that any generic algorithm g can be used to approximate $\langle\!\langle E(y) \rangle\!\rangle_{A^i}$, provided that g is *sound* and *generative*.

We first introduce a notion of *i-witnessed assignment A*: if a variable y has a witness J with $\delta(J) \leq i$, then y has a witness in an *i-witnessed assignment A*.

Definition 5 (i-witnessed). For a given environment E, and an assignment A for E, we say that A is i-witnessed if:

$$\forall y \in Vars(E).\ ([\![y]\!]_E \cap \mathcal{J}^i) \neq \emptyset \ \Rightarrow\ A(y) \neq \emptyset$$

Generativity of g means that, if A is i-witnessed, then the assignment computed using g is $(i{+}1)$-witnessed, so that, by repeated application of g starting from A^0, every non-empty variable will be eventually "witnessed" (Property 2).

Hereafter, we say that a triple (S, E, A) is coherent if E is guarded and closing for S, and if $Vars(E) = Vars(A)$.

Definition 6 (Soundness of g). A function $g(\_, \_)$ mapping each pair assertion-assignment to a set of JSON values is *sound* iff, for every coherent (S, E, A), if A is sound for E, then $g(S, A) \subseteq [\![S]\!]_E$.

Definition 7 (Generativity of g). A function $g(\_, \_)$ mapping each pair assertion-assignment to a set of JSON values is *generative* for an assertion S iff for any E and A such that (S, E, A) is coherent:

(1) if $([\![S]\!]_E \cap \mathcal{J}^1) \neq \emptyset$, then $g(S, A) \neq \emptyset$;
(2) for any $i \geq 1$, if A is i-witnessed, and if $([\![S]\!]_E \cap \mathcal{J}^{i+1}) \neq \emptyset$, then $g(S, A) \neq \emptyset$.

g is *generative for E* if it is generative for $E(y)$ for each y.

We can now define our bottom-up algorithm (Algorithm 1).

Prepare(E) rewrites E and prepares all the extra variables needed for generation, as explained later. Then, we initialize A^0 as the empty assignment $\lambda y.\ \emptyset$. We repeatedly execute a pass that sets $A^i(y) = \mathsf{Gen}(E(y), A^{i-1})$ for any y such that $A^{i-1}(y) = \emptyset$ — we call it "pass i". We say that a pass i is *useful* if there exists y such that

Algorithm 1: Bottom-up witness generation

```
1  BottomUpGenerate(x,E)
2      Prepare (E);
3      ∀y. A[y] := nextA[y] := ∅ ;
4      while A[x] == ∅ do
5          for y in vars(E) where A[y] == ∅ do
6              nextA[y] := Gen(E(y),A)
7          if (∀y. nextA[y] == A[y]) then return (unsatisfiable);
8          else
9              ∀y. A[y] := nextA[y];
10     return (A[x]);
```

$A^i(y) \neq \emptyset$ while $A^{i-1}(y) = \emptyset$, and we say that pass i was *useless* otherwise. Before each pass i, if $\langle\langle x \rangle\rangle_{A^{i-1}} \neq \emptyset$, then the algorithm stops with success. After pass i, if the pass was useless, the algorithm stops with "unsatisfiable".

Property 2 (Soundness and completeness). *If* Gen *is sound and is generative for E after preparation, then Algorithm 1 enjoys the following properties.*

(1) If the algorithm terminates with success after step i, then $A^i(x)$ is not empty and is a subset of $[[x]]_E$.
(2) If the algorithm terminates with "unsat.", then $[[x]]_E = \emptyset$.
(3) The algorithm terminates after at most $|Vars(E)| + 1$ passes.

Proof sketch. Property (1) is immediate. For (2), we first prove the following property: if the algorithm terminates with "unsatisfiable" after step j, then, for every variable y: $A^j(y) = \emptyset \implies [[y]]_E = \emptyset$. Assume, towards a contradiction, that there is a non empty set of variables Y such that $y \in Y \implies (A^j(y) = \emptyset \wedge [[y]]_E \neq \emptyset)$. Let d be the minimum depth of $\bigcup_{y \in Y} [[y]]_E$, and let w be a variable in Y and such that d is the minimum depth of the values in $[[w]]_E$. Minimality of d implies that every variable z with a value in $[[z]]_E$ whose depth is less than $d-1$ has a witness in A^j, hence, since the step j was useless, every such z has a witness in A^{j-1}, hence A^{j-1} is $(d-1)$-witnessed, hence, by generativity, w should have a witness generated during step j, which contradicts the hypothesis.

If the algorithm terminates with "unsatisfiable", this means that $\langle\langle x \rangle\rangle_{A^{j-1}} = \emptyset$, hence $\langle\langle x \rangle\rangle_{A^j} = \emptyset$ since the step j was useless, hence $[[x]]_E = \emptyset$, since we proved that $A^j(y) = \emptyset \implies [[y]]_E = \emptyset$.

Property (3) holds since at every useful pass the number of variables such that $A^i(y) \neq \emptyset$ diminishes by at least 1. □

We finally describe the phases of preparation and generation for object groups, corresponding to the functions `Prepare` and `Gen` of Algorithm 1, respectively. For reasons of space we leave the description of preparation and generation for arrays in the full paper [9], where we also detail generation for strings and numbers.

Preparation is a crucial phase, where we make explicit the interactions between different object or array operators, and we create new variables to manage these interactions.

6.3 Object group preparation and generation

6.3.1 Constraints and requirements. We say that an assertion $S = \text{props}(r : x)$ or $S = \text{pro}_0^M$ is a *constraint*. A *constraint* has the following features: (a) $\{\ \} \in [[S]]_E$ and (b) $\{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E \implies \{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E$ — constraints can prevent the addition of members, but they never require the presence of a member, similarly to a *for all fields* quantifier.

We say that an assertion $S = \text{pattReq}(r : x)$ or $S = \text{pro}_m^\infty$ with $m > 0$ is a *requirement*. A *requirement S* has the following features: (a) $\{\ \} \notin [[S]]_E$ and (b) $\{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E \implies \{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E$ — requirements can require the addition of a member, but they never prevent adding a member, similarly to an *exists field* quantifier.

6.3.2 Preparation and generation. For a typical object group, where every pattern is trivial and where each type in each pattReq is just $x_\mathbf{t}$ (which we use to indicate the only variable whose body is **t**), object generation is very easy. Consider the following group:

$$\{\ \text{type(Obj)}, \text{props}(\text{"}a\text{"} : x), \text{pattReq}(\text{"}a\text{"} : x_\mathbf{t}), \text{pattReq}(\text{"}c\text{"} : x_\mathbf{t})\ \}$$

In order to generate a witness, we just need to generate a member $k : J$ for each required key, respecting the corresponding props constraint if present. Hence, here we generate a member $\text{"}a\text{"} : J$ where $J \in A^i(x)$, and a member $\text{"}c\text{"} : J'$, where J' is arbitrary.

Unfortunately, in the general case where we have non-trivial patterns and where the pattReq operator specifies a non-trivial schema for the required member, the situation is much more complex, and we must keep into account the following issues:

(1) need to compute the intersections between patterns of different assertions;
(2) need to generate new variables when patterns intersect;
(3) possibility for one member to satisfy many requirements.

To exemplify the first two problems, consider the following object group: $\{\ \text{type(Obj)}, \text{props}(p : x), \text{pattReq}(r : y), \text{pro}_1^1\ \}$.

There are two distinct ways of producing a witness $\{\ k : J\ \}$ for the object above: either we generate a k that matches $r \sqcap \overline{p}$, and a witness J for y, or we generate a k that matches $r \sqcap p$, and a witness J for $x \wedge y$. This exemplifies the first two issues above:

(1) patterns: we need to compute which of the combinations $r \sqcap \overline{p}$ and $r \sqcap p$ have a non-empty language, in order to know which approaches are viable w.r.t. to pattern combination;
(2) new variables: we need a new variable whose body is $x \wedge y$, in order to generate a witness for this conjunctive schema.

Let us say that a member $k : J$ has shape $r : S$ when $k \in L(r)$ and J is a witness for S. Then, we can rephrase the example above by saying that an object $\{\ k : J\ \}$ satisfies that object group iff $k : J$ either has shape $(r \sqcap \overline{p} : y)$ or $(r \sqcap p : x \wedge y)$.

To exemplify the last problem — one member possibly satisfying many requirements — consider the following object group:

$$\{\ \text{type(Obj)}, \text{pattReq}(r_1 : y_1), \text{pattReq}(r_2 : y_2), \text{pro}_{min}^{Max}\}$$

In order to satisfy both requirements, we have two possibilities:

(1) producing just one member with shape $r_1 \sqcap r_2 : y_1 \wedge y_2$;
(2) producing two members, with shapes $r_1 : y_1$ and $r_2 : y_2$.

In order to explore all possible ways of generating a witness, we need to consider both possibilities. But, in order to consider the first possibility, we need a new variable whose body is $y_1 \wedge y_2$.

We solve all these issues by transforming, during the preparation phase, every object into a form where all possible interactions between assertions are made explicit, and we create a fresh new variable for every conjunction of variables that is relevant for witness generation.

6.3.3 Object group preparation. Consider a generic object group

$$\{\ \mathsf{type(Obj)},\quad \mathsf{props}(p_1 : x_1), \ldots, \mathsf{props}(p_m : x_m),\ \mathsf{pattReq}(r_1 : y_1), \ldots, \mathsf{pattReq}(r_n : y_n), \mathsf{pro}^{Max}_{min}\ \}$$

We use CP (*constraining part*) to denote the set of props assertions $\{\!|\mathsf{props}(p_i : x_i) \mid i \in 1..m|\!\}$ and RP (*requiring part*) to denote the set of pattReq assertions. Any witness for this object group is a collection of fields (k, J) where every field satisfies every constraint $\mathsf{props}(p_i : x_i)$ such that $k \in L(p_i)$, and such that every requirement $\mathsf{pattReq}(r_j : y_j)$ is satisfied by a matching field. Hence, every field is associated to a set $CP' \subseteq CP$ of constraints and to a set $RP' \subseteq RP$ of requirements. Only some pairs of sets (CP', RP') make sense, because of pattern compatibility. Object preparation generates all, and only, the pairs (actually, the *triples*, as we will see) that will be useful to the task of exploring all ways of generating a witness.

Formally, to every pair (CP', RP'), where $CP' \subseteq CP$ and $RP' \subseteq RP$, we associate a *characteristic pattern* $cp(CP', RP')$ that describes all strings (maybe none) that match every pattern in (CP', RP') and no pattern in $(CP \setminus CP', RP \setminus RP')$, as follows.

Definition 8 (Characteristic pattern). Given an object group $\{\mathsf{type(Obj)}, CP, RP, \mathsf{pro}^{Max}_{min}\}$ and two subsets $CP' \subseteq CP$ and $RP' \subseteq RP$, the characteristic pattern $cp(CP', RP')$ is defined as follows:

$$\begin{aligned} &cp(CP', RP') \\ &\quad = \quad (\textstyle\bigsqcap_{\mathsf{props}(p:\_)\in CP'} \overline{p}\,) \sqcap (\textstyle\bigsqcap_{\mathsf{props}(p:\_)\in (CP\setminus CP')} \overline{p}\,) \\ &\qquad (\textstyle\bigsqcap_{(\mathsf{pattReq}(r:\_)\in RP'} \overline{r}\,) \sqcap (\textstyle\bigsqcap_{(\mathsf{pattReq}(r:\_)\in (RP\setminus RP')} \overline{r}\,) \end{aligned}$$

Consider for example the following object group, corresponding, modulo variable names, to a fragment of our running example (Figure 1(d)):

$$\{\mathsf{type(Obj)}, \mathsf{props}(\text{"}b\text{"} : x), \mathsf{pattReq}(\text{"}a\text{"} : y1), \mathsf{pattReq}(\text{"}a.*\text{"} : y2)\}$$

For space reason, we adopt the following abbreviations for the assertions that belong to CP and RP:

$$pb = \mathsf{props}(\text{"}b\text{"} : x), \qquad ra = \mathsf{pattReq}(\text{"}a\text{"} : y1),$$
$$ras = \mathsf{pattReq}(\text{"}a.*\text{"} : y2)$$

Here we have 2^3 pairs (CP', RP') that are elementwise included in (CP, RP), each pair defining its own characteristic pattern; for each pattern we indicate an equivalent extended regular expression (".+" stands for any non-empty string) or $\emptyset$ when the pattern has an empty language:

$$\begin{aligned} cp(\{\!|\}, \{\!|\}) &= \overline{b} \sqcap \overline{a} \sqcap \overline{a.*} &&\equiv \overline{b} \sqcap \overline{a.*} \\ cp(\{\!|\}, \{\!|ra|\!\}) &= \overline{b} \sqcap a \sqcap \overline{a.*} &&\equiv \emptyset \\ cp(\{\!|\}, \{\!|ras|\!\}) &= \overline{b} \sqcap \overline{a} \sqcap a.* &&\equiv a.+ \\ cp(\{\!|\}, \{\!|ra, ras|\!\}) &= \overline{b} \sqcap a \sqcap a.* &&\equiv a \\ cp(\{\!|pb|\!\}, \{\!|\}) &= b \sqcap \overline{a} \sqcap \overline{a.*} &&\equiv b \\ cp(\{\!|pb|\!\}, \{\!|ra|\!\}) &= b \sqcap a \sqcap \overline{a.*} &&\equiv \emptyset \\ cp(\{\!|pb|\!\}, \{\!|ras|\!\}) &= b \sqcap \overline{a} \sqcap a.* &&\equiv \emptyset \\ cp(\{\!|pb|\!\}, \{\!|ra, ras|\!\}) &= b \sqcap a \sqcap a.* &&\equiv \emptyset \end{aligned}$$

All different pairs (CP', RP') define languages that are mutually disjoint by construction, but many of these are empty, as in this example. The non-empty languages cover all strings, by construction, hence they always define a partition of the set of all strings.

Consider now a member $k : J$ which we may use to build a witness of the object group. The key k matches exactly one non-empty characteristic pattern $cp(CP', RP')$, hence J must be a witness for all variables x_i such that $\mathsf{props}(p_i : x_i) \in CP'$, but, as far as the assertions $\mathsf{pattReq}(r_j : y_j) \in RP'$ are concerned, there is much more choice. If J is a witness for every such y_j, then this member satisfies all requirements in RP'. But it may be the case that some of these y_j's are mutually exclusive, hence we must choose which ones will be satisfied by J. Or, maybe, none of the y_j is satisfied by J, but we may still use $k : J$ in order to satisfy a $\mathsf{pro}^{\infty}_{m}$ requirement with $m \neq 0$. Hence, in order to explore all different ways of generating a member $(k : J)$ for a witness of the object group, we must choose a pattern $cp(CP', RP')$, and a subset RP'' of RP' that we require J to satisfy. Hence, we define a *choice* to be a triple (CP', RP', RP''), with $RP'' \subseteq RP'$. The $(CP', RP', \_)$ part specifies the pattern that is satisfied by k, while the $(CP', \_, RP'')$ part, with $RP'' \subseteq RP'$, specifies the variables that J must satisfy.

We also distinguish *R-choices*, where RP'' is not empty, hence they are useful in order to satisfy some requirements in RP, and *non-R-choices*, where RP'' is empty, hence they can only be used to satisfy a $\mathsf{pro}^{\infty}_{m}$ requirement. The only choices that may describe a member are those where $L(cp(CP', RP'))$ is not empty; we call them *non-cp-empty choices*.

Definition 9 (Choice, R-Choice, cp-empty choice). Given an object group $\{\ \mathsf{type(Obj)}, CP, RP, \mathsf{pro}^{M}_{m}\ \}$ with constraining part $CP = \{\!|\mathsf{props}(p_i : x_i) \mid i \in 1..m|\!\}$ and $RP = \{\!|\mathsf{pattReq}(r_j : y_j) \mid j \in 1..n|\!\}$, a choice is a triple (CP', RP', RP'') such that $CP' \subseteq CP$, $RP'' \subseteq RP' \subseteq RP$. The *characteristic pattern* $cp(CP', RP', RP'')$ of a choice is defined by its first two components, as follows:

$$cp(CP', RP', RP'') = cp(CP', RP')$$

The *schema* of the choice $s(CP', RP', RP'')$ is defined by the first and the third component, as follows:

$$s(CP', RP', RP'') \;=\; \bigwedge_{\mathsf{props}(p:x)\in CP'} x \wedge \bigwedge_{\mathsf{pattReq}(r:y)\in RP''} y$$

A choice is *cp-empty* if $L(cp(CP', RP', RP''))$ is empty, is *non-cp-empty* otherwise.

A choice is an *R-choice* if $RP'' \neq \{\!|\}$, is a *non-R-choice* otherwise.

In the object group of our previous example we have 4 non-cp-empty pairs, $(\{\!|\}, \{\!|\})$, $(\{\!|pb|\!\}, \{\!|\})$, $(\{\!|\}, \{\!|ras|\!\})$, $(\{\!|\}, \{\!|ra, ras|\!\})$, which correspond to the following 8 non-cp-empty choices – for each, we indicate the corresponding schema.

$$\begin{aligned} s(\{\!|\}, \{\!|\}, \{\!|\}) &= x_{\mathbf{t}} && \text{non-R-choice} \\ s(\{\!|pb|\!\}, \{\!|\}, \{\!|\}) &= x && \text{non-R-choice} \\ s(\{\!|\}, \{\!|ras|\!\}, \{\!|\}) &= x_{\mathbf{t}} && \text{non-R-choice} \\ s(\{\!|\}, \{\!|ras|\!\}, \{\!|ras|\!\}) &= y2 && \text{R-choice} \\ s(\{\!|\}, \{\!|ra, ras|\!\}, \{\!|\}) &= x_{\mathbf{t}} && \text{non-R-choice} \\ s(\{\!|\}, \{\!|ra, ras|\!\}, \{\!|ra|\!\}) &= y1 && \text{R-choice} \\ s(\{\!|\}, \{\!|ra, ras|\!\}, \{\!|ras|\!\}) &= y2 && \text{R-choice} \\ s(\{\!|\}, \{\!|ra, ras|\!\}, \{\!|ra, ras|\!\}) &= y1 \wedge y2 && \text{R-choice} \end{aligned}$$

The schema of a choice is always a conjunction of variables, say $x_1 \wedge \ldots \wedge x_n$. During bottom-up generation, we need to know which non-cp-empty choices have a witness in the current assignment A^i, hence we need to associate every non-cp-empty choice with just one variable, not with a conjunction. Hence, we need to create a new variable y for each conjunction $x_1 \wedge \ldots \wedge x_n$ that we have never

seen before, then we execute GDNF normalization over $x_1 \wedge \ldots \wedge x_n$, transforming it into a guarded disjunction of typed groups S, then we add $y : S$ to the current environment and we apply *preparation* again to this new variable; we call this process *and-completion*. In the example above, this may be the case for $y1 \wedge y2$, unless $y1 \wedge y2$ is Boolean-equivalent to some variable that already exists.

Preparation can be regarded as a sophisticated form of and-elimination. Here, *and-completion* plays the same role that not-completion plays for not-elimination: it creates the new variables that we need in order to push conjunction through the object group operators. But, crucially, and-completion is *lazy*: we do not pre-compute every possible conjunction, but only those that are really needed by some specific non-cp-empty choice. This laziness is crucial for the practical feasibility of the algorithm: when different constraints, or requirements, are associated to disjoint patterns, we have very few non-cp-empty choices, and in most cases they do not need any fresh variable, as in the example. Despite laziness, this prepare-generate-normalize-prepare loop can still generate a huge number of variables. We keep their number under control using the ROBDDTab data structure that we introduced in Section 5.1, which allows us to create a new variable only when none of the existing variables is boolean-equivalent to its body; this crucial optimization also ensures that this phase can never generate an infinite loop.

Hence, object preparation proceeds as follows:

(1) determine the set of non-cp-empty pairs (CP', RP'), that is the pairs such that $cp(CP', RP')$ is not empty;
(2) for each non-cp-empty pair (CP', RP') compute the corresponding choices (CP', RP', RP'') and, if the variable intersection $vi = s(CP', RP', RP'')$ has no equivalent variable in the environment, add a new variable $x : s(CP', RP', RP'')$ to the environment, apply GDNF reduction to vi, apply preparation to the GDNF-reduced conjunction.

Step (1) has, in the worst case, an exponential cost, but in practice it is much cheaper: in the common case where every pattern matches a single string, a set of n properties and requirements generates at most $n + 1$ non-empty pairs (one for each string plus one for the complement of the string set), n R-choices, and $n + 1$ non-R-choices. Since before preparation we have at most $O(N)$ distinct variables (where N is the input size), step (2) may generate at most $O(2^N)$ new variables, each of which has a body which can be prepared in time $O(2^{poly(N)})$. Hence, the global cost of this phase is still $O(2^{poly(N)})$. In our implementation we use an algorithm, sketched in the full paper [9], that runs in polynomial time in the common case when the number of non-cp-empty pairs is actually polynomial in the size of the object group, and our experiments show that this cost is, for most real-world schemas, tolerable.

Property 3. *Object preparation can be performed in $O(2^{poly(N)})$ time.*

6.3.4 Witness generation from a prepared object group. After the object group has been prepared once for all, at each pass of bottom-up witness generation we use the following sound and generative algorithm, listed as Algorithm 2, to compute a witness for the prepared object group starting from the current assignment A^i.

In a nutshell, we (1) pick a list of choices that contains enough R-choices to satisfy all requirements — each choice will correspond to one field in the generated object, and vice versa; (2) we verify that the list is *pattern-viable*, i.e., that it does not require two fields with the same name; (3) to satisfy any unfulfilled pro_m^∞ requirement, we add some non-R-choices, still keeping the choice list *pattern-viable*, as defined above. In order to keep the search space in $O(2^{poly(N)})$, we limit ourselves to the subset of the *disjoint* solutions, and we prove that it is big enough to have a complete algorithm.

In greater detail, consider a generic object group with the form $\{\ \text{type(Obj)}, CP, RP, \text{pro}_m^M\ \}$ and assume that the corresponding non-cp-empty choices have been prepared.

To generate an object, we first choose a list of choices that satisfies all of RP. To reduce the search space, we first observe that a single object can be described by many different choice lists. For example, assume that '1' belongs to both $[\![x]\!]_E$ and $[\![y]\!]_E$ and assume that:

$$rx = \text{pattReq}(\text{"}a|b\text{"} : x)$$
$$ry = \text{pattReq}(\text{"}a|b\text{"} : y)$$
$$RP = \{\ rx, ry\ \}$$

then $\{\ \text{"}a\text{"} : 1, \text{"}b\text{"} : 1\ \}$ is described by each the following four choice lists (and by others), where every choice could be used to generate/describe each of the two members:

$$CL_1 = [\ (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\}), \quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|ry|\!\})\ ]$$
$$CL_2 = [\ (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), \quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|\}|\!\})\ ]$$
$$CL_3 = [\ (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), \quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\})\ ]$$
$$CL_4 = [\ (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), \quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\})\ ]$$

This example shows that we do not need to explore any possible choice list, but just *enough* choice lists to generate *all* witnesses. To this aim, we focus on *disjoint solutions*, defined as follows, whose completeness will be proved in Theorem 12.

Definition 10 (Disjoint solution, Minimal disjoint solution). Fixed a set RP, a size limit M, and a set of choices $\mathbb{C}$, a multiset $\mathbb{C}' = \{\!|(C_l, R'_l, R''_l) \mid l \in L|\!\}$ with elements in $\mathbb{C}$ is a *solution* iff:

$$\bigcup_{l \in L} R''_l = RP \ \text{ and } \ |\mathbb{C}'| \leq M$$

The solution is *disjoint* if: $i \neq j \implies R''_i \cap R''_j = \emptyset$.

The solution is *minimal* if every choice in $\mathbb{C}'$ is an R-choice.

In the previous example, only CL_1 and CL_2 are disjoint, and only CL_1 is disjoint and minimal.

Object generation depends on the current assignment A^i. We say that a variable x is *Witnessed* (in A^i) when $A^i(x) \neq \emptyset$, and is *NonWitnessed* otherwise. We say that a choice is *Witnessed*, or *NonWitnessed*, when its schema variable is *Witnessed*, or is *NonWitnessed*. In order to generate a witness, we first generate a *disjoint minimal solution* for RP with bound M, only using R-choices that are *Witnessed*. Then, in order to deal with the constraint that all names in an object are distinct, we check that the solution is *pattern-viable*. Informally, pattern-viability ensures that, if we have n choices in the solution with the same characteristic pattern cp, then the language of cp has at least n different strings, which can be used to build n different members corresponding to those n choices. We will exemplify the issue after the definition.

Definition 11. A set of choices $\mathbb{C}$ is pattern-viable iff for every pair (CP', RP'), the number of choices in $\mathbb{C}$ with shape $(CP', RP', \_)$

is smaller than the number of words in $L(cp(CP', RP'))$:

$$\forall CP', RP'.$$
$$|\{\!|(CP', RP', RP'') \mid (CP', RP', RP'') \in \mathbb{C}|\!\}| \leq |L(cp(CP', RP'))|$$

For example, the following choice list $\mathbb{C}$ is not viable since it describes an object with two members that share the same characteristic pattern "a" that only contains one string:

$$rx = \text{pattReq}(\text{"}a\text{"} : x),\ ry = \text{pattReq}(\text{"}a\text{"} : y)$$
$$\mathbb{C} = [\quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\}),\quad (\{\!|\}, \{\!|rx, ry|\!\}, \{\!|ry|\!\})\quad]$$

But it would be viable if the pattern "a" were substituted by "$a|b$".

Finally, for each viable disjoint solution, we check whether it also satisfies the pro_m^∞ requirement (line 6 of Algorithm 2). If it does not, we try and extend the solution by adding some *Witnessed* non-R-choices (line 7). Observe that the disjoint solution contains each R-choice (CP', RP', RP'') at most once, because of disjointness; however, we can add the same non-R-choice as many times as we need in order to reach m members. A non-R-choice C can only be added if the result remains viable; hence, a minimal disjoint solution $\mathbb{C}$ may have a viable extension $\mathbb{C}'$ of length m, obtained by adding a multiset of non-R-choices (lines 6-13), or it may not have such a viable extension, and then we need to start from a different minimal solution. If no viable disjoint solution admits a viable extension of length at least m, then the algorithm returns "no witness" (according to the current assignment). Otherwise, we use the extended solution $\mathbb{C}'$ to build a witness: for each choice $C \in \mathbb{C}'$, we generate a name k satisfying $cp(C)$, we pick a value J from $A^i(var(C))$, and the set of members $k : J$ that we obtain is a witness for the object group. When n different choices inside $\mathbb{C}'$ have the same characteristic pattern, we generate n different names, which is always possible since the solution is viable.

Algorithm 2: Object witness generation

```
1  Gen(RPart, WitRChoices,WitNonRChoices, min, Max,)
2      for Solution in minDisjointSols (WitRChoices,RPart,Max) do
3          if (viable(Solution)) then
4              missing := min − size(Solution);
5              nonViableChoices := ∅;
6              while (missing > 0 and nonViableChoices !=WitNonRChoices)
                 do
7                  choose NRC from (WitNonRChoices-nonViableChoices);
8                  if (viable([NRC]++Solution)) then
9                      Solution := [NRC]++Solution;
10                     missing := missing-1;
11                 else nonViableChoices := [NRC]++nonViableChoices;
12             if (missing == 0) then
13                 return ("Witnessed", WitnessFrom(Solution));
14     return ("NonWitnessed");
```

Theorem 12 (Soundness and generativity). *Algorithm* Gen *is sound and generative.*

Proof sketch. Proving soundness is trivial, as our algorithm is sound by construction. For generativity, assume that the group $S = \{\ \text{type}(\text{Obj}), CP, RP, \text{pro}_{min}^{Max}\ \}$ has a witness of depth $d + 1$ in $[\![S]\!]_E$. Assume that A is d-witnessed for E. We want to prove that Gen, applied to S and A, will generate at least one witness. Let $J = \{a_1 : J_1, \ldots, a_l : J_l\}$ be a witness for S in E with depth $d + 1$. We can extract from the fields of J a multiset of choices $\mathbb{C} = (C_i', R_i', R_i'')$ with $i \in \{1..l\}$, that describes these fields, as detailed in [9]. We prove that all these choices are *Witnessed* in A, by exploiting the fact that J has depth $d + 1$, hence every J_i that appears in the witness has depth d at most, and A is d-witnessed. Finally, we prove that our algorithm would generate at least one solution for the group. To this aim, we first remove every non-R-choice from $\mathbb{C}$, hence obtaining a minimal disjoint solution, and we then add non-R-choices back if required by a pro_m^∞ requirement, and we observe that this is a viable solution, hence our algorithm would find it. □

Property 4 (Complexity). *Given a schema of size N, each run of the* Gen *algorithm has a complexity in* $O(2^{poly(N)})$.

6.4 Completeness and correctness

The algorithm described in this paper is correct and complete.

Theorem 13 (Correctness and completeness). *The witness generation algorithm is correct and complete: it returns a witness if, and only if, the schema admits a witness, and otherwise it indicates that the schema is not satisfiable*

Proof sketch. This follows from Property 1, Property 2, and Theorem 12 (more details in [9]).

□

7 EXPERIMENTAL ANALYSIS

7.1 Implementation and experimental setup

We implemented our algorithm in Java 11, employing the Brics library [24] to generate witnesses from patterns, and the *jdd* library [29] for ROBDDs. Our experiments were run on a virtualized machine deployed on a server with a 12-core Intel Xeon Silver 2.40GhZ CPU, 64 GB of RAM, running Debian GNU/Linux 11. Witnesses were validated by an external tool [2], and additionally by hand, since the external tool reported false negatives in a few cases. Each schema is processed by a single thread, and all reported times are measured for a single run. Our reproduction package [3] can be used to confirm our results.

7.2 Tools for comparative experiments

Due to the lack of equivalent tools, we compare our tool against a Data Generator and a Containment Checker.

Data generator (DG). We use an open source test data generator for JSON Schema [16] (version 0.4.6). This Java implementation pursues a try-and-fail approach: an example is first generated, then validated against the schema, and potentially refined if validation fails, exploiting the error message. This tool lends itself to a comparison although it is not able to detect schema emptiness: given an unsatisfiable schema, it will always return an (invalid) instance.

Containment checker (CC). We compare our tool against the containment checker by Habib et al. [20] (version 0.0.5), described in [22], and designed to check interoperability of data transformation operators [15].

7.3 Schema collections

We conduct experiments with different schema collections. Table 1 states the respective numbers of satisfiable/unsatisfiable schemas.

Real-world schemas. For the GitHub collection, we retrieved virtually all files from GitHub that present the features of JSON Schema, based on a BigQuery search on the GitHub public dataset. We downloaded the 80K identified schemas (shared online [13]). We performed duplicate-elimination and data cleaning (see [9]), arriving at 6,427 schemas, 40 of which are unsatisfiable (according to our tool and confirmed by direct inspection). We renamed all occurrences of `uniqueItems`, treating it as a user-defined keyword.

The three remaining real-world collections correspond to specifications of standards for deploying applications (Kubernetes [23]), ruling interactions within a specific system (Snowplow [4]), and describing data produced by content management systems (Washington Post [27]). To increase the number of processable schemas, we inlined references to external schemas. An earlier version of these collections where already used in [22] to check inclusion. Almost all schemas are satisfiable, except 5 from Kubernetes.

Hand-written schemas. Real-world schemas reflect real usage, and can be quite big, but they focus on the commonest operators and combination of operators. Hence, for stress-testing we inserted in our reproduction packages 235 handwritten schemas that are small but have been crafted to exemplify complex interactions between the language operators. To illustrate such an interaction, consider the following schema.

$$\{\, r : \mathsf{props}(a : x) \wedge \mathsf{props}(a.* : y) \wedge \mathsf{req}(a),\\ x : \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(a(c|e)),\\ y : \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(a(b|c)) \,\}$$

Here we have an interaction between two props and a req with overlapping patterns, and associated with two different variables x and y whose schemas present non-trivial overlapping.

Array operators also present interactions, as in the following example.

$$\{\, r : \mathsf{item}(1 : x) \wedge \mathsf{cont}_1^1(y),\\ x : \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{cont}_2^\infty(t),\\ y : \mathsf{cont}_1^\infty(\mathsf{type}(\mathsf{Num}) \wedge \mathsf{mulOf}(3))\}$$

This example describes an array with schema r that contains another array with schema $x \wedge y$, this one having at least two elements (because of $\mathsf{cont}_2^\infty(t)$), one of which is multiple of 3.

The collection has been built by systematically considering operators for objects, arrays, strings and numbers, following software engineering principles for testing complex programs. Ultimately, this collection has proved particularly helpful in debugging.

Synthesized schemas. We include schemas that are neither real-world nor hand-written, but they are *synthesized*, that is, they are generated from the reference test suite for JSON Schema validation [25], designed to cover all language operators. The derivation is described in [5, 6], and yields triples (S_1, S_2, b) where the Boolean b specifies whether $S_1 \subseteq S_2$ holds for schemas S_1, S_2. Here, we restrict ourselves to schemas in Draft-04, since the CC tool is restricted to this version. We excluded selected schemas that contain features that we do not yet support, such as the `format` keyword (a mere technicality) or references to external files.

We check a containment $S_1 \subseteq S_2$ by trying to generate a witness for the schema $S_1 \wedge \neg S_2$, which is unsatisfiable if, and only if, $S_1 \subseteq S_2$ holds; we thus obtain both satisfiable and unsatisfiable schemas. The CC tool accepts two schemas as input and does not need this encoding. We also test the DG tool, where comparison is only meaningful for pairs where $S_1 \wedge \neg S_2$ is satisfiable, since the DG tool cannot recognize unsatisfiable schemas.

7.4 Research hypotheses

We test the following hypotheses: (H1) *correctness* of our implementation, that we test with the help of an external tool that verifies the generated witnesses; (H2) *completeness* of our implementation, that we test by using an ample and diverse test-set; (H3) it can be used to fulfill some specific tasks better than existing tools; (H4) it can be implemented to run in *acceptable time* on sizable real-world schemas, despite its asymptotic complexity. We test the latest hypothesis by applying our tool to a vast set of real-world schemas.

7.5 Experimental results

7.5.1 Correctness and completeness. When testing each tool, we distinguish four outcomes: *success*, when a result is returned and it is correct; *failure*: when the code raises a run-time error or a timeout, that we set to 3,600 secs; *logical error on satisfiable schema*, when the input schema S is satisfiable but the code returns either "unsatisfiable" or a witness that does not actually satisfy S; *logical error on unsatisfiable schema*, when the input schema is unsatisfiable but a presumed witness is nevertheless returned.

We summarize the results of the experiments in Table 1. Our tool produces no logical error in any of our schema collections. With the GitHub schemas, it fails with "timeout" for 0.44% of schemas (28 schemas), with a "ref-expansion" for 0.01% (1 schema), and with "out of memory", when calling the automata library, for 0.36% of schemas (23 schemas). No failures arise in the other schema collections, supporting hypothesis H1.

The DG tool successfully handles 94.20% of the GitHub schemas, and has similar correctness ratio for the other real-world schemas but it performs poorly regarding correctness on handwritten schemas, and cannot be really used for inclusion checking, since it does not detect unsatisfiability. It is difficult to compare run-times between tools. Essentially, on most schemas the two tools have comparable times, evident when looking at the median times, but there is a small percentage of files where our tool takes a very long time, and this is reflected on our disproportionately high average time.

The synthesized schemas show that our tool supports a much wider range of language features (hypothesis H2), which is natural since the CC tool targets a language subset, while completeness is core to our work.

We can conclude that our tool advances the state-of-the-art for containment checking and witness generation, especially for schemas that present aspects of complexity (hypothesis H3).

7.5.2 Runtime on real-world schemas. We next test hypothesis H4. In each of the three biggest collections, 95% of the files are elaborated in less than 4.0 secs, with median ≤65 ms, and average ≤5 secs. The smaller Washington Post collection presents higher times, which will be discussed in Section 7.6. These results are coherent with hypothesis H4.

Table 1: Schema collections, correctness and completeness results, median/95th percentile/average runtime (in seconds).

Collection	#Total	#Sat/ #Unsat	Size (KB) Avg/Max	Tool	Success	Failure	Errors sat.	Errors unsat.	Med. Time	95% -tile	Avg. Time
GitHub	6,427	6,387/40	8.7/1,145	Ours	99.19%	0.81%	0%	0%	0.019 s	0.749 s	4.289 s
				DG	94.2%	2.86%	2.43%	0.51%	0.021 s	0.082 s	0.190 s
Kubernetes	1,092	1,087/5	24.0/1,310.7	Ours	100%	0%	0%	0%	0.013 s	0.510 s	0.577 s
				DG	99.54%	0%	0%	0.46%	0.023 s	0.069 s	0.031 s
Snowplow	420	420/0	3.8/54.8	Ours	99.52%	0.48%	0%	no unsat	0.065 s	3.864 s	2.071 s
				DG	94.76%	0%	5.24%	no unsat	0.024 s	0.078 s	0.032 s
WashingtonPost	125	125/0	21.1/141.7	Ours	100%	0%	0%	no unsat	0.042 s	132.690 s	23.349 s
				DG	96.8%	0%	3.2%	no unsat	0.030 s	0.079 s	0.042 s
Handwritten	235	197/38	0.1/109.4	Ours	100%	0%	0%	0%	0.070 s	3.063 s	2.593 s
				DG	8.51%	34.04%	49.36%	8.09%	0.023 s	0.132 s	0.049 s
Containment-draft4	1,331	450/881	0.5/2.9	Ours	100%	0%	0%	0%	0.004 s	0.038 s	0.011 s
				DG	28.78%	30.88%	0.07%	40.27%	0.020 s	0.034 s	0.019 s
				CC	35.91%	62.96%	0.15%	0.98%	0.003 s	0.096 s	0.036 s

7.6 Qualitative Insights

Several interesting insights can be extracted from an analysis of the size-time relationship for the GitHub collection, represented by the scatterplot in Figure 3. The histograms at the top and at the right-hand side indicate that schema size and run-time are distributed along 6 orders of magnitude, with a strong concentration on the low part of both axes, which forced us to use a log-log scale. In the log-log plot, we observe a cloud with a slope of about 1, suggesting a linear correlation, but we also observe that every file-size exhibits many outliers, and that long-running schemas can be found everywhere along the file-size axis. This clearly indicates that the runtime is affected more by the presence of specific combinations of operators, which may take little space but cause exponential runtime, than by schema size.

Indeed, our complexity analysis shows that exponential complexity is triggered by some specific operations, such as (1) object preparation, when different patterns overlap, requiring the generation of an exponential number of *choices* and of new variables; (2) reduction to DNF; and (3) pattern manipulation.

We tried to complement this theoretical knowledge with observations on the data. We applied data-mining techniques to correlate features of the schemas with the run-time. The feature that correlates more clearly with very long run-time is the presence of a `"maxLength"`: n statement with $n > 65,000$, which induces the creation of a large automaton. Other features with a strong correlation with high run-time are the presence of `"enum"` with extremely long lists of arguments, that may then cause the generation of very big terms during DNF reduction, and of `"oneOf"` with long lists of arguments, which again can generate big terms during DNF.

The Washington Post collection requires a specific analysis to explain its high 95% percentile time and average time. It is a smallish collection (125 schemas), where 20% of the files require around 130 secs for their elaboration, while all the others require less than 1 sec, with a global median of 42 ms. All the "slow" files are very similar, with more than 2K nodes in their syntax trees. By selectively deleting specific subtrees, we concluded that the high time is due to pattern overlapping between an instance of `"patternProperties"` and a corresponding instance of `"properties"`, confirming our theoretical knowledge of the strong influence of pattern overlapping over the complexity of object preparation. The small number of files in this collection and their high homogeneity explains the anomaly of the result.

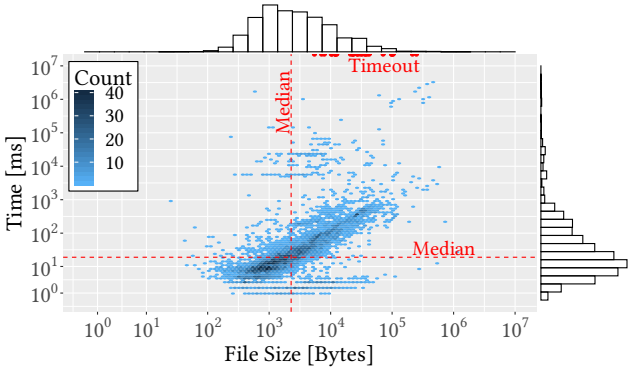

Figure 3: File size vs. runtime for GitHub schemas; log-log scatterplot with histograms, highlighting the medians. Top right, the sizes of the files causing timeouts are shown.

8 CONCLUSIONS

In this paper we have described an algorithm for witness generation, designed for the specific features of JSON Schema object and array operators. Our extensive experimental evaluation proves the practical viability of the approach, and provides insight into the actual behavior of the algorithm on real-world schemas.

We have left the implementation of the `uniqueItems` operator out of the scope of the current paper in order to keep the size and complexity of this work under control, but the fundamental techniques that we have designed, for object and array preparation and generation, still apply, with some important generalizations that we believe deserve a dedicated analysis.

Another possible development is to move from "witness generation", where the only goal is to generate any witness that proves satisfiability, to "example generation", where we generate a set of witnesses designed to satisfy some criterion of "completeness" or "realism", for applications ranging from schema explanation to test-set and workload generation.

ACKNOWLEDGMENTS

The research has been partially supported by the MIUR project PRIN 2017 FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems) and by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) – 385808805. We thank Stefan Klessinger for creating the reproduction package.

REFERENCES

[1] 2022. JSON Schema Faker. Available on GitHub at https://github.com/json-schema-faker/json-schema-faker and as an interactive tool at https://json-schema-faker.js.org. Retrieved 19 September 2022.

[2] 2022. JSON schema validator. https://github.com/networknt/json-schema-validator Retrieved 19 September 2022.

[3] 2022. Reproduction Package on GitHub. Temporarily available at GitHub from https://github.com/sdbs-uni-p/JSONSchemaWitnessGeneration, will be moved to Zenodo, for long-term availability.

[4] Snowplow Analytics. 2022. Iglu Central. https://github.com/snowplow/iglu-central, commit hash 726168e. Retrieved 19 September 2022.

[5] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. Reproduction package: A Test Suite for JSON Schema Containment. Available on Zenodo at https://zenodo.org/record/5336931#.YshD0XZBxD8 and maintained on GitHub at https://github.com/sdbs-uni-p/json-schema-containment-testsuite.

[6] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A Test Suite for JSON Schema Containment. In *Proc. ER 2021*. 19–24. http://ceur-ws.org/Vol-2958/paper4.pdf

[7] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. 2021. A Tool for JSON Schema Witness Generation. In *Proc. EDBT 2021*. 694–697. https://doi.org/10.5441/002/edbt.2021.86 Tool Demo.

[8] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. 2021. Un Outil de Génération de Témoins pour les schémas JSON A Tool for JSON Schema Witness Generation. In *Proc. Actes de la conférence BDA*. Informal proceedings.

[9] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. arXiv:2202.12849 [cs.DB] Accompanying technical report avilable online at https://arxiv.org/abs/2202.12849.

[10] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas And Types For JSON Data. In *Proc. EDBT*. 437–439.

[11] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *Proc. SIGMOD Conference*. 2060–2063.

[12] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Proc. Actes de la conférence BDA*. Informal proceesings, article available online at https://hal.archives-ouvertes.fr/hal-03190106/document.

[13] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A JSON Schema Corpus. A corpus of over 80thousand JSON Schema documents, collected from open source GitHub repositories, using Google BigQuery, in July 2020. Available on Zenodo (10.5281/zenodo.5141199) and maintained on GitHub (https://github.com/sdbs-uni-p/json-schema-corpus).

[14] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Negation-Closure for JSON Schema. arXiv:2202.13434 [cs.DB] Accompanying technical report, available online https://arxiv.org/abs/2202.13434.

[15] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. LALE: Consistent Automated Machine Learning, In Proc. KDD Workshop on Automation in Machine Learning (AutoML@KDD). *Computing Research Repository* abs/2007.01977. https://arxiv.org/abs/2007.01977

[16] Jim Blackler. 2022. JSON Generator. Available at https://github.com/jimblackler/jsongenerator. Retrieved 19 September 2022.

[17] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proc. PODS*. 123–135. https://doi.org/10.1145/3034786.3056120

[18] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[19] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. *Tree Automata Techniques and Applications*. 262 pages. Available online at https://hal.inria.fr/hal-03367725/file/tata.pdf.

[20] IBM Corp. 2021. jsonsubschema. https://github.com/IBM/jsonsubschema Retrieved 19 September 2022.

[21] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. 2021. New Workflows in NoSQL Schema Management. In *Proc. SEA-Data@VLDB (CEUR Workshop Proceedings)*, Vol. 2929. CEUR-WS.org, 38–39.

[22] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proc. ISSTA*. 620–632. https://doi.org/10.1145/3460319.3464796

[23] Kubernetes. 2022. Kubernetes JSON Schemas. https://github.com/instrumenta/kubernetes-json-schema, commit hash 133f848.

[24] Anders Møller. 2021. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. Available at https://www.brics.dk/automaton/. Retrieved 19 September 2022.

[25] JSON Schema Org. 2022. JSON Schema Test Suite. https://github.com/json-schema-org/JSON-Schema-Test-Suite. Retrieved 19 September 2022.

[26] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW*. 263–273. https://doi.org/10.1145/2872427.2883029

[27] The Washington Post. 2022. ans-schema. https://github.com/washingtonpost/ans-schema, commit hash abdd6c211. Retrieved 19 September 2022.

[28] Larry J. Stockmeyer. 1974. *The Complexity of Decision Problems in Automata Theory and Logic*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[29] Arash Vahidi. 2020. JDD. https://bitbucket.org/vahidi/jdd/src/master/ Retrieved 19 September 2022.

[30] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02 Retrieved 19 September 2022.

[31] A. Wright, G. Luff, and H. Andrews. 2017. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-wright-json-schema-validation-01*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-wright-json-schema-validation-01 Retrieved 19 September 2022.