Witness Generation for Classical JSON Schema

LYES ATTOUCHE, Université Paris Dauphine - PSL, Paris, France
MOHAMED-AMINE BAAZIZI, Sorbonne Université, LIP6, Paris, France
DARIO COLAZZO, LAMSADE, Université Paris Dauphine - PSL, Paris, France
GIORGIO GHELLI, Dipartimento di Informatica, Università di Pisa, Pisa, Italy
CARLO SARTIANI, Università degli Studi della Basilicata, Potenza, Italy
STEFANIE SCHERZINGER, Universität Passau, Passau, Germany

JSON Schema is an important, evolving standard schema language for families of JSON documents. It is based on a complex combination of structural and Boolean operators, including negation, as well as mutually recursive variables. The static analysis of JSON Schema documents comprises practically relevant problems, including schema satisfiability, inclusion, and equivalence. These three can be reduced to witness generation: given a schema, generate an element of the schema — if it exists — otherwise report unsatisfiability. Schema satisfiability, inclusion, and equivalence have been shown to be decidable, by reduction to reachability in alternating tree automata. However, no witness generation algorithm has yet been formally described. We contribute a first, direct algorithm for JSON Schema witness generation. We study its effectiveness and efficiency, in experiments over several schema collections, including thousands of real-world schemas. Our focus is on the completeness of the language (where we only exclude the `"uniqueItems"` operator), on the ability of the algorithm to run in reasonable time on a large set of real-world examples, despite the exponential complexity of the problem, and on proving its correctness and completeness.

CCS Concepts: • **Information systems → Database design and models**; • **Theory of computation → Type theory;**

Additional Key Words and Phrases: JSON Schema, satisfiability, inclusion

ACM Reference Format:
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2026. Witness Generation for Classical JSON Schema. *ACM Trans. Datab. Syst.* 51, 4, Article 22 (June 2026), 45 pages. https://doi.org/10.1145/3799416

This work is partly supported by the European Union under the scheme HORIZON-INFRA-2021-DEV-02-01 — Preparatory phase of new ESFRI research infrastructure projects, Grant Agreement n.101079043, "SoBigData RI PPP: SoBigData RI Preparatory Phase Project". We also acknowledge the support of the PRIN Project "BioConceptum" (2022AEEKXS), under the NRRP MUR program funded by the NextGenerationEU.
Authors' Contact Information: Lyes Attouche, Université Paris Dauphine - PSL, Paris, Île-de-France, France, e-mail: lyes.attouche@dauphine.fr; Mohamed-Amine Baazizi, Sorbonne Université, LIP6, France, e-mail: mohamed-amine.baazizi@lip6.fr; Dario Colazzo, LAMSADE, Université Paris Dauphine - PSL, Paris, Île-de-France, France, e-mail: dario.colazzo@dauphine.fr; Giorgio Ghelli, Dipartimento di Informatica, Università di Pisa, Pisa, Toscana, Italy, e-mail: ghelli@di.unipi.it; Carlo Sartiani (corresponding author), Università degli Studi della Basilicata, Potenza, Italy, e-mail: sartiani@gmail.com; Stefanie Scherzinger, Universität Passau, Passau, Bayern, Germany, e-mail: stefanie.scherzinger@uni-passau.de.

ACM 0362-5915/2026/06-ART22
https://doi.org/10.1145/3799416

1 Introduction

JSON Schema is a schema language that allows one to impose constraints on the structure and the admissible values of a family of JSON documents. In particular, JSON Schema allows users to impose constraints on objects, arrays, and primitive values, like numbers and strings; these constraints can be richly combined through traditional logical combiners (e.g., **and**, **or**, **not**, and **exclusive or**), and recursive data structures can be specified through recursive references.[1] The most important decision problems related to JSON Schema are *validation*, *satisfiability*, *inclusion*, *equivalence*, and *witness generation*. Validation is a simple problem, that has already been extensively studied. The other problems are still open, and they can all be expressed in terms of witness generation, which is the core contribution of this article. We now define them formally.

Inclusion $\mathcal{S} \subseteq \mathcal{S}'$: does, for each value J, $J \vDash \mathcal{S} \Rightarrow J \vDash \mathcal{S}'$? Checking schemas for inclusion (or containment) is of great practical importance: if the output format of a tool is specified by a schema $\mathcal{S}$, and the input format of a different tool by a schema $\mathcal{S}'$, the problem of format compatibility is equivalent to schema inclusion $\mathcal{S} \subseteq \mathcal{S}'$; given the high expressive power of JSON Schema, this "format" may include detailed information about the range of specific parameters. Schema inclusion also plays a central role in schema evolution, with questions of the kind: will a value that respects the new schema still be accepted by tools designed for legacy versions? If not, what is an example of a problematic value?

Equivalence $\mathcal{S} \equiv \mathcal{S}'$: does, for each value J, $J \vDash \mathcal{S} \Leftrightarrow J \vDash \mathcal{S}'$? Some tools exist that rewrite schemas into equivalent schemas, and equivalence checking is essential in order to guarantee the correctness of their output, as we show in Section 6.6.

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

Witness generation for $\mathcal{S}$, a constructive generalization of satisfiability: given $\mathcal{S}$, generate a value J such that $J \vDash \mathcal{S}$, or return "unsatisfiable" if no such value exists. In the first case, we call J a *witness*. Schema inclusion $\mathcal{S} \subseteq \mathcal{S}'$ can be immediately reduced to witness generation for $\mathcal{S} \wedge \neg\mathcal{S}'$: $\mathcal{S}$ is included in $\mathcal{S}'$ when every instance J that satisfies $\mathcal{S}$ also satisfies $\mathcal{S}'$, that is, when no instance satisfies both $\mathcal{S}$ and $\neg\mathcal{S}$, hence $\mathcal{S} \subseteq \neg\mathcal{S}$ holds if and only if witness generation returns "unsatisfiable" when applied to $\mathcal{S} \wedge \neg\mathcal{S}'$. Using a witness generation algorithm rather than a satisfiability algorithm has a crucial advantage: when inclusion does not hold, witness generation returns a witness J for $\mathcal{S} \wedge \neg\mathcal{S}'$ that can be used to provide users with an explanation: $\mathcal{S}$ is not included in $\mathcal{S}'$ *because* of values such as J. We can similarly solve a "witnessed" version of equivalence: given $\mathcal{S}$ and $\mathcal{S}'$, either prove that one is equivalent to the other, or provide an explicit witness J that belongs to one, but not to the other.

Open challenges. Witness generation for JSON Schema is a difficult problem. First of all, JSON Schema includes conjunction, disjunction, negation, structural operators, recursive second-order variables, and recursion under negation; this combination is quite difficult to deal with. Secondly, for each JSON type, the different structural operators have complex interactions, as in the following example, where the negated `"patternProperties"` and `"required"` force the presence of members whose names match `"^a"` and `"^abz$"` (this is explained later in the article), `"maxProperties" : 1` forces these two members to be one, and, finally, operator `"patternProperties"` forces the value of that member to satisfy the subschema *var2*, since `"abz"` also matches `"z$"`. This kind of interaction is unique, among logical languages, to JSON Schema, and it makes it impossible to consider each operator independently from the others.

[1] In this article we refer to Draft-06 of JSON Schema, which is the most advanced version of Classical JSON Schema, enriched with the operators `"minContains"` and `"maxContains"`; this choice is explained in Section 3.2.

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

Many JSON Schema operators would be sufficient to make the problem computationally intractable by themselves, as proved by Bourhis et al. [17], based on results of Pezoa et al. [33]. Their combination exacerbates the difficulty of the design of a *complete* algorithm that is *practical*, that is, of an algorithm that is correct and complete, but is also able to run in a reasonable time over the vast majority of real-world schemas.

Contributions. The main contribution of this article is an original sound and complete algorithm for checking the satisfiability of an input schema $\mathcal{S}$, generating a witness J when the schema is satisfiable. Our algorithm supports the whole language without uniqueItems, which we omitted since, as shown by Bourhis et al. [17], the complexity upper bound of satisfiability would jump to 2EXPTIME. JSON Schema **regular expressions** (**REs**) are ECMA regular expressions, whose universality is undecidable [20], which makes satisfiability and witness generation undecidable for JSON Schema. In practice, most schemas use REs that do not have this problem, since they can easily be expressed using the standard RE syntax (r in Figure 1), whose universality is decidable, hence, as detailed in Section 3.5, we restrict our study to a version of JSON Schema where REs are expressed using the standard syntax, as done in other theoretical studies about JSON Schema [17].

While the existence of an algorithm for this specific problem follows from Bourhis et al. [17], where the problem is proved to be EXPTIME-complete, we are the first to explicitly describe such 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 schema manipulations, some of which, such as *preparation*, are unique to JSON Schema, and have not been proposed before. In this article, we detail each algorithm phase, show that each is in $O(2^{poly(N)})$, and focus on preparation and generation of objects and arrays, the phases completely original to this work.

The practical applicability of our algorithm for witness generation and inclusion checking is proved by a large experimental evaluation. For witness generation, we use four real-world datasets and a handwritten dataset, engineered to test the most complex aspects of the JSON Schema language. For inclusion checking, we use a dataset synthesized from the standard schemas provided by *JSON Schema Org* [32], a handwritten dataset of 282 pairs, a dataset extracted from the test cases of a tool for the simplification of schemas [25], and a dataset of 1,056 schema pairs created from successive versions of real-world schemas from the curated SchemaStore repository [3]. Our experiments show that, despite its exponential complexity, our algorithm behaves quite well even on schemas with tens of thousands of nodes.

Witness vs. examples. In this article, we describe a generalization of satisfiability checking, where the answer "$\mathcal{S}$ is satisfiable" is enriched with an instance J that satisfies $\mathcal{S}$. We call J a *witness* of the satisfiability of $\mathcal{S}$, rather than an *example*, since example generation is often used to indicate a different problem, where the schema is regarded as satisfiable, and the focus is not on witnessing the satisfiability, but rather on the generation of many and different examples that can exemplify the different forms that instances of $\mathcal{S}$ may take. Of course, the techniques that we present in this article can be used as foundations for example generation algorithms.

Article outline. In Section 2, we analyze related work. In Section 3, we briefly describe JSON Schema and our algebraic framework. In Sections 4 and 5, we describe the algorithm, divided in

preliminary phases and final phases. In Section 6, we present an extensive experimental evaluation of our approach. In Section 7, we draw our conclusions.

2 Related Work

Overviews of schema languages for JSON have been presented by Pezoa et al. [33] and by Bourhis et al. [17]. Pezoa et al. [33] introduced the first formalization of JSON Schema and showed that it cannot be captured by MSO or tree automata because of the uniqueItems constraints. They first focused on validation and proved that it can be decided in polynomial time. They also showed that JSON Schema can simulate tree automata; hence, JSON Schema satisfiability is EXPTIME-hard. (See Suárez Barría [14] for a detailed proof of how a quantified alternating tree automaton can be encoded in polynomial time into an equivalent JSON Schema document.)

Bourhis et al. [17] 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. Since they map JSON Schema onto recursive JSL logic, and provide a specific kind of alternating tree automata for this logic, their technique suggests a possible algorithm for witness generation, based on the generation of a reachability proof for the automaton and its transformation into a corresponding witness. However, we have not found a way to design a practical algorithm starting from their proof, which is not surprising since reachability algorithms for alternating automata are designed to prove complexity upper bounds, not to be practical tools. They are typically based on the exploration of all subsets of the automaton state set [19], hence on a sequence of complex operations on a set of sets whose dimension, for the real-world schemas in our dataset, may be in the realm of $2^{10,000}$. Although exponentiality cannot be avoided in the worst case, we believe that a different approach should be pursued to design a practical algorithm.

To the best of our knowledge, the only tool currently available to check the satisfiability of a schema is the containment checker described by Habib et al. [23]. While it has been designed for schema containment checking, e.g., $\mathcal{S}_1 \subseteq \mathcal{S}_2$, it can also be exploited for schema satisfiability since $\mathcal{S}$ is satisfiable if and only if $\mathcal{S} \not\subseteq$ `false`, where `false` is the 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. Consider, for instance, the following recursive schema.

```
1 {"type" :"object",
2  "properties" :{"name" :{"type" :"string" },"surname" :{"type" :"string"},
3                 "children" :{"type" :"array", "items" :{"$ref" :"#"}} }}
```

This schema describes objects with two mandatory properties, `"name"` and `"surname"`, as well as an optional property `"children"`, whose value is a, possibly empty, array of objects conforming to the root schema. The tool by Habib et al. [23] processes references by replacing each reference with the JSON value being referenced. This strategy cannot be applied to recursive references, and, therefore, the tool by Habib et al. raise an "UnsupportedRecursiveRef" exception on this schema.

There exist several tools for data generation starting from a JSON Schema (see [16] and [2] for instance). They generate JSON data starting from a schema, but are based on a trial-and-error approach and cannot detect unsatisfiable schemas. Other tools, like the Wisconsin Benchmark Data Generator [27], do not take as input a JSON schema and, therefore, are not relevant here.

Baazizi et al. [13] discuss negation-completeness for JSON Schema. They show that negation cannot be completely eliminated in JSON Schema because of some minor issues in the way some specific operators are defined. They propose an algebraic version of JSON Schema where negation can be eliminated, and they define an algorithm for not-elimination, which we use in our algorithm. We recap the algebra and the not-elimination algorithm in Sections 3.3 and 4.2.

Table 1. Complexity of Inclusion for JSON Schema, DTDs, XSDs, and Tree Automata

Language/Formalism	Inclusion	Equivalence
DTDs	PSPACE-complete [29]	In PSPACE [29]
XML Schema	PSPACE-hard [30]	PSPACE-hard [30]
	In EXPSPACE [21]	In EXPSPACE [21]
JSON Schema (w/o `"uniqueItems"`)	EXPTIME-complete [17]	EXPTIME-complete [17]
Tree automata	EXPTIME-complete [35]	EXPTIME-complete [35]

XML is a data exchange language that can represent trees, similarly to JSON, although the two languages differ in many aspects, as discussed in detail by Bourhis et al. [17] and by Arenas et al. [8] (see Part VIII). Still, it is worth comparing the main complexity results about inclusion and equivalence for JSON Schema (without `"uniqueItems"`), DTDs, XML Schema, and tree automata, and we summarize them in Table 1.

Since JSON Schema is a logical language, it is natural to ask whether it could be mapped onto some well-studied logical languages. First-order languages, such as First Order Logic, Prolog, or Datalog, for example, are not candidates for such a mapping, since these are languages where variables are first-order, that is, they denote one domain value, and sets are defined by collecting sets of first-order variable assignments (often called *valuations*). JSON Schema is a second-order logic, where semantics is defined by a *single* assignment of each variable to a set (see Section 3.4). A more natural mapping is that of JSON Schema into a modal logic, such as the μ-calculus, since these are logics that share many of the fundamental features of JSON Schema, and this approach has been successfully followed by Bourhis et al. [17] in order to prove the complexity results that we have already described. However, the approaches used to prove satisfiability for these languages [14] cannot be reused for the design of a realistic algorithm.

Our prior work. A preliminary version of this article appeared as a conference paper [11]. Compared to that conference version, here we present the following major additions:

— While in the conference version we only described preparation and generation for objects, in this article, we present the same phases for arrays as well, whose treatment is significantly different.
— While in the conference version we omitted most proofs and sketched the most important ones, here we provide the theoretical machinery underneath our approach and the non trivial proofs that are needed to explain the subtle points of our algorithm, and to prove its correctness.
— Differently from the reduced version, here we provide a detailed complexity analysis.
— Finally, we have made substantial extensions to the experimental analysis. In particular, we added experiments about the use of our implementation for inclusion checking, using both handwritten and real-world schemas, and about its use for ensuring the correctness of the results of a tool for schema and-merging.

3 JSON Schema and the Algebra

We first introduce the JSON data model, then JSON Schema, and finally the algebraic syntax that we adopt to manipulate JSON Schema.

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. Objects represent sets of members, each member being a name-value pair, where no name can be present twice, and arrays represent ordered sequences of values.

$J ::=$	$B \mid O \mid A$		**JSON expressions**
$B ::=$	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 (Value equality). 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; hence, objects which only differ in the order of the members are equal.

3.2 JSON Schema

Many successive versions of the JSON Schema standard have been published, notably Draft-06 of April 2017 [40], Draft 2019-09 of September 2019 [38], and Draft 2020-12 of December 2020 [39]. Draft 2019-09 changed the evaluation model of JSON Schema (see [12] for a detailed discussion), and for this reason, it is regarded as the first Draft that defines *Modern JSON Schema*, while the previous Drafts define variations of *Classical JSON Schema* [7]. The new semantic aspects of Draft 2019-09 have not had an ample adoption up to now, hence we decided to base our work on Draft-06; however, we decided to include the operators `"minContains"` and `"maxContains"` introduced with Draft 2019-09 since they are very interesting in the context of witness generation and they do not depend on the new semantics features of Modern JSON Schema.

JSON Schema uses JSON syntax; a JSON Schema document (or *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, and the value for `"type"` is an array of strings.

A JSON Schema document (*schema*) denotes a set of JSON documents (*values/instances*) that satisfy it. The language offers the following abilities.

- Base type specification: it is possible to define complex properties of collections of base type values, such as all strings that satisfy a given ECMA regular expression (`"pattern"`), all numbers that are a multiple of a given decimal number (`"multipleOf"`) or that are included in a given interval (`"minimum"`, `"maximum"`,...).
- Array specification: it is possible to specify the types of the elements for both uniform and non-uniform arrays (`"items"`), to restrict the minimum and maximum size of the array, to bound the number of elements that satisfy a given property (`"contains"`, `"minContains"`, `"maxContains"`), and also to enforce uniqueness of the items (`"uniqueItems"`).
- Object specification: it is possible to require for certain names to be present or to be absent, to specify the schemas of both optional or mandatory members, all of this by denoting classes of names using POSIX regular expressions (`"properties"`, `"patternProperties"`, and `"required"`). It it possible to specify that some assertions depend on the presence of some members (`"dependencies"`), and it is possible to specify an interval for the number of members that are present.
- Boolean combination: one can express union, intersection, and complement of schemas (`"anyOf"`, `"allOf"`, `"not"`), and also a generalized form of mutual exclusion (`"oneOf"`).
- Mutual recursion: mutually recursive schema variables can be defined (`"definitions"`, `"$ref"`).

In the next section, we describe JSON Schema by giving its translation into a simpler algebra.

3.3 The algebra with negation and the positive algebra

JSON Schema is not *algebraic*, because the meaning of some assertions is modified by the surrounding assertions, which makes formal manipulation very difficult. Moreover, the language is rich in

$$
\begin{array}{lll}
m \in \mathsf{Num}^{-\infty}, M \in \mathsf{Num}^{\infty}, l \in \mathbb{N}_{>0}, i \in \mathbb{N}, j \in \mathbb{N}^{\infty}, q \in \mathsf{Num}, k \in \mathsf{Str} & & \\
T & ::= & \mathsf{Arr} \mid \mathsf{Obj} \mid \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Num} \\
r & ::= & \emptyset \mid \epsilon \mid \mathit{char} \mid (r|r) \mid r \cdot r \mid (r)^* \\
e & ::= & r \mid \Sigma_i^j \mid \overline{e} \mid e_1 \sqcap e_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \\
S & ::= & \mathsf{ifBoolThen}(b) \mid \mathsf{pattern}(e) \mid \mathsf{betw}_m^M \mid \mathsf{xBetw}_m^M \mid \mathsf{mulOf}(q) \mid \mathsf{props}(e : S) \mid \mathsf{req}(k) \mid \mathsf{pro}_i^j \\
 & & \mid \mathsf{item}(l : S) \mid \mathsf{items}(i^+ : S) \mid \mathsf{cont}_i^j(S) \mid \mathsf{type}(T) \mid\ S_1 \wedge S_2 \mid S_1 \vee S_2 \mid x \\
\text{with neg.:} & & \mid \neg S \\
\text{positive:} & & \mid \mathsf{notMulOf}(q) \mid \mathsf{pattReq}(e : S) \mid \mathsf{contAfter}(i^+ : S) \\
E & ::= & x_1 : S_1, \ldots, x_n : S_n \\
D & ::= & S \text{ defs } (E)
\end{array}
$$

Fig. 1. Syntax of the *with-negation* and *positive* algebras.

redundant operators, such as `"if" – "then" – "else"` and `"dependencies"`, which can both be easily translated in terms of `"not"` and `"anyOf"`.

For these reasons, in our implementation, we translate JSON Schema onto an *algebra*, that is just an algebraic version of JSON Schema with less redundant operators.

This algebra is a simplified version of the one presented by Baazizi et al. [13]. It is defined by a subset of JSON Schema operators, big enough so that all the others can be translated into this algebra in a very simple way. In contrast to JSON Schema, the algebra possesses the property of substitutability. This implies that any schema can be substituted in any context by another schema that validates the same instances. Furthermore, it is expressed in a non-JSON syntax, primarily for the purposes of substitutability and compactness. The algebra is similar to the recursive JSL logic defined by Bourhis et al. [17], but has a different aim; recursive JSL logic is a tool for theoretic research, where minimality and elegance are very important, while this algebra is an implementation tool, hence a greater degree of adherence to JSON Schema is more important than minimality.

The first step of our approach is the translation of an input schema into our algebraic representation (Section 3.6), and the second step is not-elimination (Section 4.2). For the first step, we use an *algebra with negation*. For not-elimination, we use a *positive algebra* where we remove negation $\neg S$, but we add three new operators: $\mathsf{notMulOf}(n)$, $\mathsf{pattReq}(e : S)$, and $\mathsf{contAfter}(i^+ : S)$. These algebras extend standard regular expressions with length-limitation Σ_i^j, external intersection $e_1 \sqcap e_2$ and external complement $\overline{e}$; this extension is discussed in Section 3.5. The syntax of the two algebras, *with-negation* and *positive*, which are expressive enough to capture all JSON Schema assertions of Draft-06, plus the extra operators `"minContains"` and `"maxContains"` of Draft 2019-09, is presented in Figure 1, where $\mathsf{Num}^{-\infty}$ 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 algebra with negation strictly correspond to those of JSON Schema, and in particular to their implicative semantics. The exact relationship between this algebra and JSON Schema is discussed in Section 3.6.

3.4 Semantics of the Algebras

Informally, a JSON instance J satisfies an assertion S if and only if:

— $\mathsf{ifBoolThen}(b)$: *if* the instance J is a boolean, *then* $J = b$.

— $\mathsf{pattern}(e)$: *if* J is a string, *then* J matches e.

$$
\begin{array}{lcl}
[\![\mathsf{props}(e:S)]\!]^p_E & = & \{\, J \mid \forall n \geq 0, J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \forall i \in \{1..n\}.\ k_i \in L(e) \ \Rightarrow J_i \in [\![S]\!]^p_E \,\} \\
[\![\mathsf{req}(k)]\!]^p_E & = & \{\, J \mid \forall n \geq 0, J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \exists i \in \{1..n\}.\ k_i = k \,\} \\
[\![\mathsf{item}(l:S)]\!]^p_E & = & \{\, J \mid \forall n \geq 0, J = [J_1, \ldots, J_n] \Rightarrow n \geq l \Rightarrow J_l \in [\![S]\!]^p_E \,\} \\
[\![\mathsf{items}(i^+:S)]\!]^p_E & = & \{\, J \mid \forall n \geq 0, J = [J_1, \ldots, J_n] \Rightarrow \forall j \in \{1..n\}.\ j > i \Rightarrow J_j \in [\![S]\!]^p_E \,\} \\
[\![\mathsf{cont}^j_i(S)]\!]^p_E & = & \{\, J \mid \forall n \geq 0, J = [J_1, \ldots, J_n] \Rightarrow i \leq \ |\{\, l \mid J_l \in [\![S]\!]^p_E \,\}| \ \leq j \,\}
\end{array}
$$

$$
\begin{array}{llllll}
[\![S_1 \wedge S_2]\!]^p_E & = & [\![S_1]\!]^p_E \cap [\![S_2]\!]^p_E & [\![S_1 \vee S_2]\!]^p_E & = & [\![S_1]\!]^p_E \cup [\![S_2]\!]^p_E \qquad [\![\neg S]\!]^p_E = \mathcal{J}Val(*) \setminus [\![S]\!]^p_E \\
[\![x]\!]^{p+1}_E & = & [\![E(x)]\!]^p_E & [\![x]\!]^0_E & = & \emptyset \qquad\qquad\qquad\qquad [\![S]\!]_E = \bigcup_{i \in \mathbb{N}} \bigcap_{p \geq i} [\![S]\!]^p_E
\end{array}
$$

Fig. 2. Semantics of the algebra with explicit negation: some equations.

— betw^M_m: *if* J is a number, *then* $m \leq J \leq M$; xBetw^M_m is the same but with $m < J < M$.
— $\mathsf{mulOf}(q)$: *if* J is a number, *then* $J = q \times i$ for some integer i; q is any decimal number.
— $\mathsf{props}(e : S)$ *if* J is an object and *if* (k, J') is a member of J where k matches the pattern e, *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 e.
— $\mathsf{req}(k)$: *if* J is an object, *then* it contains at least one member whose name is k.
— pro^j_i: *if* J is an object, *then* it has between i and j members.
— $\mathsf{item}(l : S)$: *if* J is an array $[J_1, \ldots, J_n]$ $(n \geq 0)$ and *if* $l \leq n$, *then* J_l satisfies S. Hence, it is satisfied by any J that is not an array and by any array that is strictly shorter than l, such as the empty array: 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 of length less than or equal to i.
— $\mathsf{cont}^j_i(S)$: *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).
— $S_1 \wedge S_2$: both S_1 and S_2 are satisfied.
— $S_1 \vee S_2$: either S_1, or S_2, or both, are satisfied.
— x: every schema is associated to an environment E, where $E = x_1 : S_1, \ldots, x_n : S_n$, is a function mapping each x_i to the schema S_i; if x is found inside a schema S that is associated to an environment E, then J satisfies x if J satisfies the schema $E(x)$.
— $\neg S$: S is not satisfied.
— $\mathsf{notMulOf}(n)$: *if* J is a number, *then* it is not a multiple of n.
— $\mathsf{pattReq}(e : S)$: *if* J is an object, *then* it contains at least one member (k, J') where k matches e 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 according to the environment $E = x_1 : S_1, \ldots, x_n : S_n$; the definitions in E are mutually recursive.

More formally, in Figure 2, we provide semantic equations for some operators. 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. For example, in "$x : (\mathsf{props}(e : y) \wedge z)$", x directly depends on z, but not on y, since y is in the scope of $\mathsf{props}(\_)$. 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. Implicative Typed Operators, therefore, play a crucial and structural role in guarding recursion. Guarded recursion is crucial, since it ensures structural progression when schema definitions are traversed by algorithms for validation and, as we will see, witness generation. The same holds for denotational semantics, whose definition is well-founded thanks to the absence of unguarded recursion.

Hereafter, we will often use the derived operators $\mathbf{t}$, $\mathbf{f}$, and $S_1 \Rightarrow S_2$; $\mathbf{t}$ is satisfied by every J, and can be expressed, for example, as pro_0^∞; $\mathbf{f}$ is satisfied by no J, and can be expressed, for example, as $\neg\mathbf{t}$; $S_1 \Rightarrow S_2$ is defined as $(\neg S_1) \vee S_2$.

The semantics of the algebras is formally defined by a function $[\![\_]\!]_\_$ that maps each pair (S, E), where S is a schema and E is a guarded environment that defines all variables in E, to the set $[\![S]\!]_E$ of the JSON instances that *satisfy* S with respect to E. The function is very similar to that described by Baazizi et al. [13], hence we report here only a few equations (see Figure 2).

Hereafter, $L(e)$ denotes the regular language generated by e. $\mathcal{J}Val(*)$ is the set of all JSON values. The set $\{1..0\}$ is empty, and universal quantification on an empty set (as in the first line of Figure 2 when $n = 0$) is true. The definitions of Figure 2 can be read as follows (ignoring the index p for a moment): the semantics of $\mathrm{props}(e : S)$ specifies that $J \in [\![\mathrm{props}(e : S)]\!]_E \Leftrightarrow$ *if* J is an object, *if* $(k_i : J_i)$ is a member where k_i matches e, *then* $J_i \in [\![S]\!]_E$.

The index p of $[\![S]\!]_E^p$ is used since otherwise the definition $[\![x]\!]_E = [\![E(x)]\!]_E$ would not be inductive, because $E(x)$ is in general bigger than x, while the definition $[\![x]\!]_E^{p+1} = [\![E(x)]\!]_E^p$ is inductive on the lexicographic pair $(p, |S|)$. The semantics $[\![S]\!]_E$ is then defined as the forall-exists limit of $[\![S]\!]_E^p$, by stipulating that an instance J belongs to the limit $[\![S]\!]_E$ if an i exists such that J belongs to every interpretation that comes after i:[2] $[\![S]\!]_E = \bigcup_{i\in\mathbb{N}} \bigcap_{p\geq i} [\![S]\!]_E^p$.

JSON Schema specifications declare that x is the same as $E(x)$ for all schemas where such interpretation never creates a loop (i.e., for all guarded schemas) and describes, verbally, the equations that we wrote, in the form without the index. Hence, Theorems 3 and 4 below prove that the semantics of Figure 2 exactly captures the official JSON Schema semantics. Their proofs, and the full set of the semantic equations, are in the Supplementary Material.

We remind here the reader that, in accordance with JSON Schema specification, in the rest of the article we assume that recursion is guarded in every schema that is analyzed by our algorithm.

Definition 2. An environment $E = x_1 : S_1, \ldots, x_n : S_n$ is *guarded* if recursion is guarded in E.

Theorem 3. *For any E guarded, the following equality holds:* $[\![E(x)]\!]_E = [\![x]\!]_E$.

Theorem 4. *For each equivalence in Figure 2, the equivalence still holds if we substitute every occurrence of* $[\![S]\!]_E^p$ *with* $[\![S]\!]_E$.

3.5 About Regular Expressions

As we specified in the Introduction, we restrict our study to a version of JSON Schema where patterns are expressions r that use the standard RE syntax, rather than the ECMA syntax. In the implementation, we map JSON Schema ECMA REs onto standard REs, using a simple incomplete algorithm. When the algorithm fails, we raise a failure. Otherwise, we use the Brics Automaton library [31] in order to generate witnesses for the translated REs. This approach is coherent with the theoretical treatment, and allows us to manage a vast majority of real-world schemas.

While we only admit standard REs in the source schema, we use a form of ***externally extended REs*** (**EEREs**) e in the algebra. Our EEREs (e) are standard REs (r) extended with three extra

[2]One cannot just set $[\![S]\!]_E = \bigcup_{p\in\mathbb{N}} [\![S]\!]_E^p$, since, because of negation, the sequence $[\![S]\!]_E^p$ is not necessarily monotonic in p; for example, if we have a definition $y : \neg(x)$, then $[\![x]\!]_E^0 = \emptyset$, hence $[\![y]\!]_E^0$ contains the entire $\mathcal{J}Val(*)$.

operators, length limitation (Σ_i^j, denoting the set of all strings whose length is included between i and j), intersection $e_1 \sqcap e_2$, and complement $\overline{e}$, that can be applied *to* REs but cannot be used *inside* the RE, so that we can write for example $\overline{r^*}$ but we cannot write $(\overline{r})^*$. We need EERE to formalize the manipulations that we perform on the algebra, such as not-elimination and preparation.

$$r \quad ::= \quad \emptyset \mid \epsilon \mid char \mid (r|r) \mid r{\cdot}r \mid (r)^* \qquad e \quad ::= \quad r \mid \Sigma_i^j \mid \overline{e} \mid e_1 \sqcap e_2.$$

This extension does not affect the expressive power of regular expressions, since Σ_i^j denotes a regular language and since regular languages are closed under intersection and complement, but affects their succinctness, hence the complexity of problems such as emptiness checking. We are going to exploit this expressive power in many different ways:

(1) to translate `"minLength"` : i as pattern(Σ_i^∞) and `"maxLength"` : j as pattern(Σ_0^j);
(2) to translate `"additionalProperties"` : $\mathcal{S}$ as props($\overline{(r_1|\ldots|r_m)}$: $\langle\mathcal{S}\rangle$), where $\overline{e}$ is applied to the standard RE $(r_1|\ldots|r_m)$ (Section 3.6);
(3) in order to translate `"propertyNames"` : $\mathcal{S}$, as detailed in the Supplementary Material;
(4) during not-elimination (Section 4.2), where pattern($\overline{r}$) is used to rewrite $\neg$pattern(r);
(5) during object preparation (Section 5.3.2), where we must express the intersection and the difference of patterns that appear in props($e : S$) and pattReq($e : S$) operators.

During the final phases of our algorithm (Section 5.3), we need to solve the *i-enumeration* problem (which generalizes emptiness) for our EEREs: for a given EERE e and for a given i, either return i words that belong to $L(e)$, or return "impossible" if $|L(e)| < i$. It is well-known that emptiness of REs extended (internally) with negation and intersection is non-elementary [36]. However, for our external-only extension, i-enumeration and emptiness can be solved in time $O(i^2 \times 2^{|e|})$, as shown in the Supplementary Material.

3.6 From JSON Schema to the Algebra

We describe here how to translate a JSON Schema schema into the corresponding algebraic expression. Hereafter, we use $\underline{k}$ to denote a pattern that only matches k when k is a string. For space reasons, we omit here the translation of `"propertyNames"` : $\mathcal{S}$, `const`, `enum`, and `dependencies`, which can be found in the Supplementary Material.

A JSON Schema schema is a JSON object whose members are assertions. Essentially, the translation $\langle\mathcal{S}\rangle$ of a schema $\mathcal{S}$ applies some simple rules to the single assertions, and combines them by conjunction, as follows:

$$\begin{array}{lcl} \langle\texttt{"multipleOf"} : q\rangle & = & \mathsf{mulOf}(q) \\ \langle\{\texttt{"a}_1\texttt{"} : \mathcal{S}_1, \ldots, \texttt{"a}_n\texttt{"} : \mathcal{S}_n\}\rangle & = & \langle\texttt{"a}_1\texttt{"} : \mathcal{S}_1\rangle \wedge \ldots \wedge \langle\texttt{"a}_n\texttt{"} : \mathcal{S}_n\rangle. \end{array}$$

However, there are some exceptions, that we describe in this section. We first describe how we map the complex referencing mechanism of JSON Schema into the simpler S defs (E) construct. We then describe the translation of `oneOf` into the algebra. Then, we describe the non-algebraic JSON Schema operators, where a group of related operators must be translated together, and we finish with the easy cases.

3.6.1 Representing Definitions and References. JSON Schema defines a `$ref` : *path* operator that allows any subschema of the current schema to be referenced, as well as any subschema of a different schema that is reachable through a URI, hence implementing a powerful form of mutual recursion. The path *path* may navigate through the nodes of a schema document by traversing its structure, or may retrieve a subdocument on the basis of a `$id` member, which associates a name to the surrounding schema object. In the most common case the subschemas that are referred are

```
(a) {"properties": {"Country": {"type": "string"},
                    "City": {"$ref": "#/properties/Country" }}}
(b) {"properties": {"Country": {"type": "string" },
                     "City": {"$ref": "#/definitions/properties_Country"}},
    "definitions": {"properties_Country": {"type": "string" }}}
```

Fig. 3. A simple schema and its normalization.

just those that are collected inside the value of a top-level `definitions` member. In this case, we just use the natural translation:

$$\langle\{a_1 : \mathcal{S}_1, \ldots, a_n : \mathcal{S}_n, \texttt{"definitions"} : \{x_1 : \mathcal{S}'_1, \ldots, x_m : \mathcal{S}'_m\}\}\rangle$$
$$= \langle\{a_1 : \mathcal{S}_1, \ldots, a_n : \mathcal{S}_n\}\rangle \text{ defs } (x_1 : \langle\mathcal{S}'_1\rangle, \ldots, x_m : \langle\mathcal{S}'_m\rangle).$$

In the general case, we collect all paths that are used in any reference assertion `$ref` : *path* and that are different from `#/definitions/`k, we retrieve the referred subschema and copy it inside the `"definitions"` member where we give it a name *name*, and we substitute all occurrences of `$ref` : *path* with `$ref` : `"#/definitions/name"`, until all paths have that format.

Example 1. We consider the JSON Schema document shown in Figure 3(a). Definition normalization produces the schema in Figure 3(b), where the target of `"$ref" : "#/properties/Country"` has been copied in the `"definitions"` section. The schema is then translated as:

$$\text{props}(\textit{Country} : \text{type}(\text{Str})) \wedge \text{props}(\textit{City} : \textit{properties\_Country}) \text{ defs}(\textit{properties\_Country} : \text{type}(\text{Str})).$$

3.6.2 Translation of `oneOf`. The assertion `"oneOf"` : $[\mathcal{S}_1, \ldots, \mathcal{S}_n]$ requires that J satisfies one of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and violates all the others. It is translated as follows, where the x_i's are fresh variables:

$$\bigvee_{i\in\{1..n\}}(\neg x_1 \wedge \ldots \wedge \neg x_{i-1} \wedge x_i \wedge \neg x_{i+1} \wedge \ldots \wedge \neg x_n) \text{ defs } (x_1 : \langle\mathcal{S}_1\rangle, \ldots, x_n : \langle\mathcal{S}_n\rangle).$$

Fresh variables are used to avoid that a single subschema is copied many times, which may cause an exponential size increase. The outermost $\bigvee$ has size $O(n^2)$, hence this encoding may still cause a quadratic size increase; this increase can be avoided using a more sophisticated linear encoding described by Baazizi et al. [13].

3.6.3 The Remaining Assertions. While most JSON Schema assertions can be translated one by one, there are four families of assertions whose semantics depends on the occurrence of other assertions of the same family as members of the same schema. These families are:

(1) `if`, `then`, `else`;
(2) `additionalProperties`, `properties`, `patternProperties`;
(3) `additionalItems`, `items`;
(4) in Draft 2019-09: `minContains`, `maxContains`, `contains`.

When translating a schema object, we first partition it into families, we complete each family by adding the predefined default value for missing operators (for example, a missing `else` becomes `"else" : true`), and we then translate each family as we specify below. All other assertions are just translated one by one.

The assertion group `"if"` : $\mathcal{S}_1$, `"then"` : $\mathcal{S}_2$, `"else"` : $\mathcal{S}_3$ is translated as follows, where $x : \langle\mathcal{S}_1\rangle$ is inserted in order to avoid duplication of $\langle\mathcal{S}_1\rangle$.

$$((x \wedge \langle\mathcal{S}_2\rangle) \vee (\neg x \wedge \langle\mathcal{S}_3\rangle)) \text{ defs } (x : \langle\mathcal{S}_1\rangle) \quad .$$

The `properties` family is translated as follows: we use $\underline{k_i}$ to denote a pattern that only matches k_i, and the pattern complement operation $\overline{e}$ to translate `additionalProperties`, which associates a schema to any name that does not match either `properties` or `patternProperties` arguments:

$$
\begin{array}{lcllcl}
\langle \texttt{"minimum"}\colon \mathrm{m}\rangle & = & \mathsf{betw}_m^{\infty} & \langle \texttt{"maximum"}\colon \mathrm{M}\rangle & = & \mathsf{betw}_0^{M} \\
\langle \texttt{"exclusiveMinimum"}\colon \mathrm{m}\rangle & = & \mathsf{xBetw}_m^{\infty} & \langle \texttt{"exclusiveMaximum"}\colon \mathrm{M}\rangle & = & \mathsf{xBetw}_{-\infty}^{M} \\
\langle \texttt{"minProperties"}\colon \mathrm{m}\rangle & = & \mathsf{pro}_m^{\infty} & \langle \texttt{"maxProperties"}\colon \mathrm{M}\rangle & = & \mathsf{pro}_0^{M} \\
\langle \texttt{"minItems"}\colon \mathrm{m}\rangle & = & \mathsf{cont}_m^{\infty}(\mathbf{t}) & \langle \texttt{"maxItems"}\colon \mathrm{M}\rangle & = & \mathsf{cont}_0^{M}(\mathbf{t}) \\
\langle \texttt{"minLength"}\colon \mathrm{m}\rangle & = & \mathsf{pattern}(\Sigma_m^{\infty}) & \langle \texttt{"maxLength"}\colon \mathrm{M}\rangle & = & \mathsf{pattern}(\Sigma_0^{M}) \\
\langle \texttt{"pattern"}\colon \mathrm{r}\rangle & = & \mathsf{pattern}(r) & \langle \texttt{"multipleOf"}\colon \mathrm{n}\rangle & = & \mathsf{mulOf}(n)
\end{array}
$$

Fig. 4. Translation rules for the simpler operators.

$$
\begin{array}{l}
\langle \texttt{"properties"} : \{k_1 : \mathcal{S}_1, \ldots, k_n : \mathcal{S}_n\}, \\
\ \texttt{"patternProperties"} : \{e_1 : \mathcal{PS}_1, \ldots, e_m : \mathcal{PS}_m\}, \\
\ \texttt{"additionalProperties"} : \mathcal{S}\rangle
\end{array}
\quad = \quad
\begin{array}{l}
\mathsf{props}(\underline{k_1} : \langle \mathcal{S}_1 \rangle) \wedge \ldots \wedge \mathsf{props}(\underline{k_n} : \langle \mathcal{S}_n \rangle) \\
\ \wedge \mathsf{props}(e_1 : \langle \mathcal{PS}_1 \rangle) \wedge \ldots \wedge \mathsf{props}(e_m : \langle \mathcal{PS}_m \rangle) \\
\ \wedge \mathsf{props}(\overline{(\underline{k_1}|\ldots|\underline{k_n}|e_1|\ldots|e_m)} : \langle \mathcal{S} \rangle)
\end{array}
$$

`items` accepts as argument either a schema $\mathcal{S}$ or an array $[\mathcal{S}_1, \ldots, \mathcal{S}_n]$; in the first case, it is equivalent to $\mathsf{items}(0^+ : \langle \mathcal{S} \rangle)$, and a co-occurring `additionalItems` is ignored, while in the second case it is equivalent to $(\mathsf{item}(1 : \langle \mathcal{S}_1 \rangle) \wedge \ldots \wedge \mathsf{item}(n : \langle \mathcal{S}_n \rangle))$, and the keyword $\texttt{"additionalItems"} : \mathcal{S}'$ means $\mathsf{items}(n^+ : \langle \mathcal{S}' \rangle)$. `"additionalItems"` without `"items"` is ignored. The family is hence translated as follows:

$$
\begin{array}{lcl}
\langle \texttt{"additionalItems"} : \mathcal{S}' \rangle & = & \mathbf{t} \\
\langle \texttt{"items"} : \mathcal{S} \rangle & = & \mathsf{items}(0^+ : \langle \mathcal{S} \rangle) \\
\langle \texttt{"items"} : \mathcal{S}, \texttt{"additionalItems"} : \mathcal{S}' \rangle & = & \mathsf{items}(0^+ : \langle \mathcal{S} \rangle) \\
\langle \texttt{"items"} : [\mathcal{S}_1, \ldots, \mathcal{S}_n] \rangle & = & (\mathsf{item}(1 : \langle \mathcal{S}_1 \rangle) \wedge \ldots \wedge \mathsf{item}(n : \langle \mathcal{S}_n \rangle)) \\
\langle \texttt{"items"} : [\mathcal{S}_1, \ldots, \mathcal{S}_n], \texttt{"additionalItems"} : \mathcal{S}' \rangle & = & (\mathsf{item}(1 : \langle \mathcal{S}_1 \rangle) \wedge \ldots \wedge \mathsf{item}(n : \langle \mathcal{S}_n \rangle)) \\
 & & \qquad \wedge \mathsf{items}(n^+ : \langle \mathcal{S}' \rangle).
\end{array}
$$

The `contains` family is translated as follows - a missing lower bound defaults to 1 (rather than the usual 0), and a missing upper bound defaults to ∞:

$$\langle \texttt{"contains"} : \mathcal{S}, \texttt{"minContains"} : m, \texttt{"maxContains"} : M \rangle = \mathsf{cont}_m^{M}(\langle \mathcal{S} \rangle).$$

Finally, all the other JSON Schema assertions are translated one by one in the natural way, as reported in Figure 4.

3.7 Complexity of the algorithm and the Small Constants Assumption

We have seen that JSON Schema can be translated to the algebra with a polynomial (actually, linear) size increase, and in the rest of the article we show that our algorithm runs in $O(2^{poly(N)})$ with respect to the size N of the input algebra, but with one important caveat: hereafter, we assume that all i and j constants different from ∞ that appear in $\mathsf{item}(i : S)$, $\mathsf{items}(i^+ : S)$, $\mathsf{contAfter}(i^+ : S)$, $\mathsf{cont}_i^j(S)$, and pro_i^j, are "small", which we formalize through the *small constants assumption*.

Definition 5 (size constants, small constants assumption). Let us define the *size constants* of a schema to be the set of all the i and j constants different from ∞ that appear in $\mathsf{item}(i : S)$, $\mathsf{items}(i^+ : S)$, $\mathsf{contAfter}(i^+ : S)$, $\mathsf{cont}_i^j(S)$, and pro_i^j. Let $\mathit{maxSizeConst}(S)$ indicate the value of the greatest size constant that appears in S.

We say that S satisfies the *small constants assumption* if $\mathit{maxSizeConst}(S) \leq |S|$.

We tested the small constants assumption on the schema corpus of Baazizi et al. [13]; the property holds for all schemas, and actually there are only 90 schemas where $\mathit{maxSizeConst}(S) \geq 0.01 \cdot |S|$. Hence, in our analysis, we will all assume that every size constant is bound by the input schema size N; whenever a result depends on this assumption, we will say that explicitly.

4 The Preliminary Phase: Not-Elimination, Stratification, DNF Reduction, Canonicalization

The generation of witnesses for JSON Schema is a difficult problem since we have to face a combination of issues: (i) presence of negation, with the extra difficulty that it can also be applied to recursive variables; (ii) presence of arbitrary conjunctions and arbitrary nesting of operators; (iii) presence of recursive variables; and (iv) complex interactions between different Implicative Typed Operators that affect the same type, such as $\mathsf{props}(e : S)$, $\mathsf{req}(k)$, and pro_i^j.

Our algorithm can be divided in two phases. In the first, preliminary, phase we reduce the schema to a specific normal form that eliminates all problems caused by negation, conjunction, and arbitrary nesting, arriving at a form where the only issues to be solved are recursion and the interactions between different structural operators; this preliminary phase is divided in four simple subphases, and it is described in this section. In the second phase, we start from this normal form and solve the remaining problems. This second phase is the core of the algorithm, where we need to introduce many original notions and techniques, and it is described in the next section.

To describe the preliminary phase, we need to introduce some terms, which are all exemplified by the schemas in Figure 5. Hereafter, for better readability, we will often use JSON Schema curly brackets $\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}$ to indicate a conjunction $S_1 \wedge \ldots \wedge S_n$.

Definition 6 (Stratified schema, Disjunctive Normal Form (DNF), Canonical Group, Canonical DNF).

- A schema is *stratified* when, for every ITO that has a schema argument S ($\mathsf{props}(e : S)$, $\mathsf{item}(l : S)$, $\mathsf{items}(i^+ : S)$, $\mathsf{cont}_i^j(S)$, $\mathsf{pattReq}(e : S)$, and $\mathsf{contAfter}(i^+ : S)$), the schema argument S is a variable; for example, the schema in Figure 5(b) is not stratified because of $\mathsf{props}(a.* : co(r) \vee x)$, while the schema in Figure 5(c) is stratified.
- A schema is in Disjunctive Normal Form (DNF) if it matches

$$\vee(\{S_{1,1}, \ldots, S_{1,n_1}\}, \ldots, \{S_{l,1}, \ldots, S_{l,n_l}\}),$$

  and no $S_{i,j}$ is a Boolean operator or a variable. For example, the schema in Figure 5(c) is not in DNF, while the one in Figure 5(d) is in DNF.
- A schema S is a *canonical group* if it is a conjunction that contains exactly one assertion $\mathsf{type}(T)$ and a set of ITOs of that same type T; for example, $\{\mathsf{type}(\mathrm{Obj}), \mathsf{props}(b : \mathrm{x})\}$, and all other conjunctions in Figure 5(e), are canonical groups.
- A schema S is in Canonical DNF when it is in DNF and every conjunction is a canonical group.

The preliminary phase generates a document S defs (E) that is positive, stratified, in DNF, and where every conjunction is canonical. To generate a witness from a non-recursive document S defs (E) that has these properties, we sort the variables in E in an order such that each variable only depends on previous variables, and we consider each variable x_i, in this order; for each variable, we consider each canonical group that appears in its DNF definition, and the variable is satisfiable if, and only if, we can generate a witness for at least one canonical group in its definition, using the witnesses of the variables that appear in that canonical group.

For example, in the schema below, we would first generate an arbitrary array J_a for x, an arbitrary number q for y, and then we may generate either an empty object or an object $\{b : J_a\}$ using the first alternative for r.

$$r\ \mathsf{defs}(x : \{\mathsf{type}(\mathrm{Arr})\}, y : \{\mathsf{type}(\mathrm{Num})\}, r : \{\mathsf{type}(\mathrm{Obj}), \mathsf{props}(b : x)\}\ \vee \{\mathsf{type}(\mathrm{Obj}), \mathsf{props}(a : y)\}).$$

This approach is based on the ability to generate a witness from an arbitrary canonical group starting from witnesses for the variables that it contains, using the algorithms described in the next

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

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

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

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

Fig. 5. (a) Original term. (b) After not-elimination. (c) After stratification, omitting unaffected variables. (d) After transformation to DNF. (e) After canonicalization.

section, and is based on the fact that the schema is positive, and that no other conjunction appears out of the canonical groups: While we are able to treat conjunction inside a canonical group, there is no general way to generate a witness for $S_1 \wedge S_2$ starting from the ability of generating witnesses for S_1 and for S_2. Positivity is needed because there is no obvious way of producing a witness for $\neg S$ starting from an arbitrary witness for S.

In this section, we describe preliminary phases of transformation in positive, stratified, and canonical DNF. These preliminary phases are illustrated by the running example of Figure 5.

Before discussing these phases, we discuss the technique of ROBDD reduction, which we employ in every phase in order to ensure termination of the algorithm and in order to make it more efficient.

4.1 ROBDD Reduction

Two schemas that only use variables and Boolean operators are *Boolean-equivalent* when they can be proved equivalent using the laws of the Boolean algebra.

As we describe in Section 5, the final phases of our algorithm define new variables which are equivalent to Boolean combinations of existing variables, then manipulate the body of these new variables, and this manipulation may recursively require the definition of new variables.

This recursive operation may loop forever, but it converges if one is able to recognize when the body of a new variable is Boolean-equivalent to the body of an existing one.

We reach this aim using **Reduced Ordered Boolean Decision Diagrams (ROBDDs)**; an ROBDD is a data structure that provides the same representation for two Boolean combinations of variables if, and only if, they are Boolean-equivalent [18]. In particular, our ROBDDTab data structure offers methods for creating fresh variables, as well as for computing the conjunction, disjunction, and negation of variables. Hence, in all phases of our algorithm, whenever we need a variable for a Boolean combination of variables S_x, we use the AND, OR, and NOT methods offered by our ROBDDTab data structure. These methods will return fresh variables only when it is strictly necessary, as pointed out before, and will reuse any variable already bound to any subexpression of S_x. As a consequence, if an expression S_y having the same ROBDD representation as S_x has already been bound to a variable y, y will be returned and no fresh variables will be created; otherwise, a fresh variable x will be associated to S_x and returned by ROBDDTab. We call this process *ROBDD reduction*. This technique makes the entire algorithm more efficient and, crucially, it ensures termination of the preparation phase (Section 5.3.2).

While in practice the cost of ROBDD construction is reasonable, the asymptotic cost of building the ROBDD representation of a Boolean expression of size n is in $O(2^n)$.

Our witness generation algorithm is in $O(2^{poly(N)})$, where N is the size of the input schema. ROBDD reduction does not increase this bound since it is applied at most $O(2^{poly(N)})$ times, which follows from the complexity of the other phases, and, crucially, as we will see in the rest of the article, from the fact that it is always applied to Boolean expressions whose size is in $O(N)$.

4.2 Not-elimination

Not-elimination (Figure 5(b)) follows the approach described in Baazizi et al. [13], and proceeds in two phases: *not-completion of variables* and *not-pushing*.

Not-completion of variables. Not-completion of variables is the operation that adds a variable not_x for every variable x as follows:[3]

$$\text{not-completion}(x_0 : S_0, \ldots, x_n : S_n) = x_0 : S_0, \ldots, x_n : S_n, not\_x_0 : \neg S_0, \ldots, not\_x_n : \neg S_n.$$

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 (and also in the *preparation* phase).

Not-pushing. We rewrite $\text{req}(k)$ as $\text{pattReq}(\underline{k} : \mathbf{t})$, and then we inductively apply not-elimination rules like the ones presented below (the full list is reported in the Supplementary Material).

$$\begin{array}{lcllcl}
\neg(\text{props}(e : S)) & = & \text{type}(\text{Obj}) \wedge \text{pattReq}(e : \neg S) & \neg(S_1 \vee S_2) & = & (\neg S_1) \wedge (\neg S_2) \\
\neg(\text{items}(i^+ : S)) & = & \text{type}(\text{Arr}) \wedge \text{contAfter}(i^+ : \neg S) & \neg(\neg S) & = & S \\
\neg(S_1 \wedge S_2) & = & (\neg S_1) \vee (\neg S_2) & \neg(x_i) & = & not_x_i.
\end{array}$$

Not-elimination can be performed in linear time and increases the schema size of a linear factor, as proved by Baazizi et al. [13].

Property 1. Not elimination on a schema S of size N can be performed in time $O(N)$, and the size of the result is $O(N)$.

Property 2. For any system where recursion is guarded, not elimination preserves the semantics of every variable.

[3]We do this, unless a variable whose body is Boolean-equivalent to $\neg S_n$ already exists, in which case that variable is used through ROBDD reduction.

4.3 Stratification

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

Stratification (Figure 5(c)) makes it easy to build a witness for a canonical group such as

$$\{\mathsf{Obj}, \mathsf{pattReq}(\texttt{"a"} : x), \mathsf{pattReq}(\texttt{"b"} : y)\},$$

after a witness has been built for x and y.

In this phase, for every ITO that has a subschema S in its syntax, such as pattReq$(\texttt{"a"} : S)$, when S is not a variable, we create a new variable $x : S$, and we substitute S with x. For every variable $x : S$ we define, we must also define its complement $not\_x : \neg S$, and perform not-elimination and stratification on $\neg S$. As specified in Section 4.1, we apply ROBDD reduction to $x : S$ and $not\_x : \neg S$.

Property 3. Stratification transforms a schema S defs (E) into a schema S' defs (E') such that $[\![S]\!]_E = [\![S']\!]_{E'}$.

Property 4. Stratification transforms a schema S defs (E) into a schema S' defs (E') such that $|S' \text{ defs } (E')|$ is in $O(N)$, where $N = |S \text{ defs } (E)|$.

Stratification can be clearly executed in linear time, but it creates new variables, hence it invokes ROBDD reduction on all the variables whose body S is a Boolean expression, whose cost is in $O(2^{|S|})$ (Section 4.1). Hence, we have the following property.

Property 5. Stratification of a schema of size N is in $O(2^N)$, including the cost of ROBDD reduction.

4.4 Transformation in Canonical DNF

A schema in DNF can be expressed, by definition, as $\vee(\wedge(S_{1,1}, \ldots, S_{1,n_1}), \ldots, \wedge(S_{l,1}, \ldots, S_{l,n_l}))$, where every $S_{i,j}$ is a **Typed Operator** (**TO**). Thanks to associativity, commutativity, and idempotency, of $\vee$ and $\wedge$, every DNF schema can be represented by a set of sets of TOs, which we call its set-of-sets representation: $\{\{S_{1,1}, \ldots, S_{1,n_1}\}, \ldots, \{(S_{l,1}, \ldots, S_{l,n_l}\}\}$.

To transform an environment E into a corresponding E^G in DNF (Figure 5(d)), we apply the following function $dnf(S)$ to the body of each variable definition. We first apply the function to the set Y_0 of the variables that do not unguardedly depend on any other variable, and then to the variables in Y_1 that unguardedly depend on variables in Y_0 only, and so on, which is possible since recursion is guarded: this specific order ensures that, when $dnf(y)$ is computed, the schema $E^G(y)$ has already been computed.

$$\begin{array}{lllll}
(1) & dnf(S) & = & \{\{S\}\} & \text{if } S \text{ is a TO} \\
(2) & dnf(y) & = & E^G(y) & \\
(3) & dnf(S_1 \vee S_2) & = & dnf(S_1) \cup dnf(S_2) & \\
(4) & dnf(S_1 \wedge S_2) & = & \{\, c_1 \cup c_2 \mid (c_1, c_2) \in dnf(S_1) \times dnf(S_2) \,\}. &
\end{array}$$

The function $dnf(\_)$ is guaranteed to terminate since recursion is guarded and, thanks to the first rule, guarded variables are not expanded. The function preserves the semantics: rules (1) and (3) transform a TO and a union into their set-of-sets representation, rule (2) substitutes a variable with its definition, and rule (4) distributes conjunction over disjunction.

This phase combines two operations each having a potentially exponential size-increasing effect — recursive expansion of variables and reduction to DNF. The phase is quite expensive, and it is actually the most expensive phase of our algorithm, according to our measures (Section 6) — observe, in Figure 5(d), how the size of rcx is the product of the sizes of r and that of $co(x)$. The

set-of-sets representation ensures that the asymptotic cost of this phase is inside $O(2^N)$, and the same upper bound applies to the size of the schema "x defs (E^G)" that is produced by this phase.

Property 6. For a given schema x defs (E), such that $N = |x \text{ defs } (E)|$, the size of x defs (E^G) is in $O(2^N)$, and it can be built in time $O(2^N)$.

Proof Sketch. Let $\mathcal{T}$ denote the set of all TOs that appear in E as subterms of $E(y)$ for any y; for example, if $E = x : (\mathsf{type(Num)} \wedge \mathsf{pattReq}(\hat{}a\$: x)) \vee \mathsf{mulOf}(3)$, then

$$\mathcal{T} = \{\mathsf{type(Num)},\ \mathsf{pattReq}(\hat{}a\$: x),\ \mathsf{mulOf}(3)\}.$$

The set-of-sets representation of the body of each variable is an element of $\mathcal{P}(\mathcal{P}(\mathcal{T}))$, since the function $dnf(S)$ does not create any TO that was not already in $\mathcal{T}$. The set $\mathcal{P}(\mathcal{P}(\mathcal{T}))$ has size $2^{(2^N)}$, hence one element of that set can be identified using $\log_2(2^{(2^N)})$ bits, that is, 2^N bits. x defs (E^G) contains at most N definitions, hence it can be represented using $N \times 2^N$ bits.

As for the construction time, the most expensive part is the computation of

$$\{\ c_1 \cup c_2\ \mid\ (c_1, c_2) \in dnf(S_1) \times dnf(S_2)\ \},$$

that may take place at most once for every $\wedge$ in the input schema, that is, at most N times. The size of $dnf(S_1) \times dnf(S_2)$ is in $O(2^N)$, the size of c_1 and c_2 is in $O(N)$, hence this computation is in $O(2^N)$.

Canonicalization. Canonicalization (Figure 5(e)) is a process defined along the lines of Habib et al. [23]. Canonicalization splits every conjunctive term of the DNF into a set of *canonical groups* (see Figure 5(e), where we also applied elementary equivalences, such as idempotence of $\vee$).

In order to transform a conjunctive term c of a DNF into a set of canonical group, we first repeatedly apply the following rewriting rules, which preserve the meaning of the conjunction. In the third rule, $ITO(T')$ are the ITOs associated to type T', which are trivially satisfied when in conjunction with a $\mathsf{type}(T)$ with $T \neq T'$:

$$\begin{array}{llll} \mathsf{type}(T), \mathsf{type}(T) & \rightarrow & \mathsf{type}(T) & \\ \mathbf{f}, S & \rightarrow & \mathbf{f} & \\ \mathsf{type}(T), \mathsf{type}(T') & \rightarrow & \mathbf{f} & T \neq T' \\ \mathsf{type}(T), S & \rightarrow & \mathsf{type}(T) & S \in ITO(T'),\ T' \neq T. \end{array}$$

The first three rules ensure that the result is either $\mathbf{f}$, which is then deleted from the disjunction, or has exactly one $\mathsf{type}(T)$ assertion, or has none. If it has exactly one $\mathsf{type}(T)$ assertion, then the fourth rule ensures that all the ITOs refer to type T. If it has no $\mathsf{type}(T)$ assertion, we transform it in the following equivalent disjunction, where $filter(\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}, T)$ is the conjunction of those ITOs in $\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}$ whose type is T:

$$(\mathsf{type(Null)}) \quad \vee \quad (\mathsf{type(Bool)} \wedge filter(c, \mathsf{Bool})) \quad \vee \quad (\mathsf{type(Str)} \wedge filter(c, \mathsf{Str})) \ldots$$

so that every conjunction in the transformed DNF denotes a set of values of the same type.

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

Property 7 (Equivalence). The phases of not-elimination, stratification, and transformation into Canonical DNF transform a JSON Schema document into an equivalent one.

Canonicalization has a linear cost and only causes a linear increase in the size, in the worst case, hence we have the following property.

Property 8 (Asymptotic cost and size expansion). The phases of not-elimination, stratification, and transformation into Canonical DNF, applied to a document of size N, have a combined cost in $O(2^N)$ and the size of the result is bounded by $O(2^N)$. The number of variables of the produced schema is bounded by $O(N)$.

$$
\begin{array}{lcl}
\langle\!\langle x \rangle\!\rangle_A & = & A(x) \\
\langle\!\langle \mathsf{ifBoolThen}(b) \rangle\!\rangle_A & = & \{\, J \mid J \in \mathcal{J}\mathit{Val}(\mathsf{Bool}) \Rightarrow J = b \,\} \\
\langle\!\langle \mathsf{props}(e : S) \rangle\!\rangle_A & = & \{\, J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \forall i \in \{1..n\}.\, k_i \in L(e) \;\Rightarrow J_i \in \langle\!\langle S \rangle\!\rangle_A \,\} \\
\langle\!\langle S_1 \wedge S_2 \rangle\!\rangle_A & = & \langle\!\langle S_1 \rangle\!\rangle_A \cap \langle\!\langle S_2 \rangle\!\rangle_A \\
\ldots & &
\end{array}
$$

Fig. 6. Rules for assignment-evaluation.

5 Preparation and Witness Generation

After the preliminary phases have produced a canonical and stratified DNF schema, we are left with the final phases of *preparation* and *bottom-up witness generation*, which are the core of our algorithm and which take care of the problems caused by recursion and, crucially, by the interactions between the different ITOs. Preparation "prepares" an object or array group by splitting and merging the different ITOs of the group in a way that makes their interactions explicit and, crucially, defines new variables when this is necessary in order to merge two ITOs. Generation uses a bottom-up approach to generate a witness from a prepared object group or array group starting from witnesses produced for the variables that appear inside the group.

We first prove that bottom-up generation is correct and complete for the JSON Schema semantics described in Section 3.4. Bottom-up generation may produce infinitely many witnesses for each variable at each pass; in Section 5.2, we prove that this can be safely approximated by a finite generator, provided that it is *sound* and *generative*. Then, we show how we prepare an object group for generation, and then introduce a sound and generative generation algorithm for prepared object groups. Finally, we show the same results for array groups.

5.1 Assignments and Bottom-up Semantics

JSON Schema semantics associates schemas with infinite sets of values. Our implementation explores a finite subset of that infinite set and stops when it cannot make any further progress, and we must prove that this finite exploration is nevertheless complete, in the sense that it generates at least one witness for any satisfiable schema.

To this aim, we first show that the bottom-up semantics that guides our implementation is equivalent to the standard semantics that we described in Section 3.4. In this section, we only consider positive schemas, which are schemas that do not use the $\neg S$ operator (Section 3.3), since witness generation is applied after not-elimination. The equivalence between top-down descriptions and bottom-up generation is a classic result in the field of logic programming [4].

Definition 7 (Assignments, Soundness, Order). 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 $y \in \mathit{Vars}(E)$: $A(y) \subseteq [\![y]\!]_E$.

We say that $A \leq A'$ iff $\forall y.\, A(y) \subseteq A'(y)$.

Fixed an environment E and an assignment A for E, we define the *assignment-evaluation* of a schema S whose variables are defined by E using the rules in Figure 6, which are the same rules that define environment-based semantics $[\![S]\!]_E$ (Figure 2), with the only difference that a variable x is not mapped to $[\![E(x)]\!]_E$ but to the set of values $A(x)$. For the schemas not containing subschemas, such as $\mathsf{ifBoolThen}(b)$, we just define $\langle\!\langle \mathsf{ifBoolThen}(b) \rangle\!\rangle_A = [\![\mathsf{ifBoolThen}(b)]\!]_E$.

We now show that, for environments in the positive algebra, we can build the semantics of a schema bottom-up, by iterated application of assignment-evaluation. We use a classical construction: we first define an assignment transformation $T_E(A)$ that plays the role of the *immediate consequence* operator used to define a bottom-up semantics for Prolog or Datalog [4]. Intuitively, if the assignment

A collects witnesses for the variables in E, then $T_E(A)$ builds new witnesses for these variables starting from the witnesses in A, using the definitions in E: the witnesses built by $T_E(A)$ derive from those in A. For example, if E contains $y : \{\mathsf{type}(\mathsf{Arr}), \mathsf{items}(0^+ : x), \mathsf{cont}_1^3(\mathbf{t})\}$, and $A(x) = \{J\}$, then $T_E(A)(y) = \{[J], [J, J], [J, J, J]\}$.

Definition 8. For a given positive environment E, the corresponding *assignment transformation* $T_E(\_)$ is the function from assignments to assignments defined as follows:

$$\forall y \in \mathit{Vars}(E).\ T_E(A)(y) = \langle\!\langle E(y) \rangle\!\rangle_A.$$

It is easy to see that, when E is positive, $A \leq A'$ implies that $T_E(A)(y) \leq T_E(A')(y)$ for every y: if A' contains more witnesses than A, then $T_E(A')$ is able to build more witnesses than $T_E(A)$. Hence, T_E is a monotone transformation of assignments, hence, it has a minimal fix-point, that, by Tarski theorem, is the limit $\mathcal{A}_E^\infty$ of the sequence $\mathcal{A}_E^i$ defined by the repeated application of T_E to the empty assignment.

Definition 9 ($\mathcal{A}_E^i$, $\mathcal{A}_E^\infty$). For a given positive environment E, the sequence of assignments $\mathcal{A}_E^i$ is defined as follows:

$$\forall y \in \mathit{Vars}(E).\ \mathcal{A}_E^0(y) = \emptyset \qquad \mathcal{A}_E^{i+1} = T_E(\mathcal{A}_E^i).$$

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

Property 9. For any positive E, the assignment $\mathcal{A}_E^\infty$ is the minimal fix-point of the assignment transformation T_E.

In Section 3.4, we sketched the schema semantics $[\![S]\!]_E$ using a top-down "denotational" style, since this style reflects how references are defined by the specifications, facilitates proving expression equivalence, and deals with negative operators. However, on positive schemas, the top-down semantics and the bottom-up fix-point coincide.

Property 10. For any positive schema S defs (E), the following equality holds: $[\![S]\!]_E = \langle\!\langle S \rangle\!\rangle_{\mathcal{A}_E^\infty}$

A final notion that we need is the *depth* $\delta(J)$ of a JSON value J, defined as the number of levels of its tree representation, as follows:

Definition 10 (Depth $\delta(J)$, $\mathfrak{J}^d$). The depth of a JSON value J, $\delta(J)$, is defined as follows, where $\max(\emptyset)$ is defined to be 0:

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

$\mathfrak{J}^d$ 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 $([\![y]\!]_E \cap \mathfrak{J}^i) \subseteq \mathcal{A}_E^i(y)$.

5.2 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.

We have defined a sequence $\mathcal{A}_E^i$ of assignments, each potentially infinite, such that $\mathcal{A}_E^\infty(x) = [\![x]\!]_E$. Our algorithm for witness generation for the schema x defs (E) produces a sequence of *finite* assignments A^i, each approximating the potentially infinite assignment $\mathcal{A}_E^i$, until it reaches either a witness for x or a situation that ensures that no witness exists.

ALGORITHM 1: Bottom-up witness generation

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

A^i is built as follows: $A^0 = \mathcal{A}^0_E$; 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 using a generation algorithm $\mathtt{Gen}(E(y), A^i)$ that computes a subset of $\langle\!\langle E(y) \rangle\!\rangle_{A^i}$; formally, $A^{i+1}(y) = \mathtt{Gen}(E(y), A^i)$. Our specific Gen algorithm is defined in the next section, but we show now that any generation algorithm *gen* can be used to compute $A^{i+1}(y)$, provided that *gen* is *sound* and *generative* (Definitions 12, 13).

We first introduce a notion of *i-complete assignment* A: if a variable y has a witness J with $\delta(J) \leq i$, then, in every *i-complete assignment* A, y has some witness (of some depth, not related with i). For example, if the minimal-depth witness of y has depth 2 and A is i-complete for any $i \geq 2$, then $A(y)$ is guaranteed not to be empty, but if A is only 1-complete, then $A(y)$ may be empty.

Definition 11 (i-complete). For a given environment E, and an assignment A for E, we say that A is i-complete if: $\forall y \in Vars(E).\ (\llbracket y \rrbracket_E \cap \mathcal{J}^i) \neq \emptyset \implies A(y) \neq \emptyset$.

Generativity of *gen* means that, if A is i-complete, then the assignment computed using *gen* is (i+1)-complete, so that, by repeated application of *gen* starting from A^0, every y with $\llbracket y \rrbracket_E \neq \emptyset$ will be eventually "witnessed" (Property 11).

Hereafter, we say that a triple (S, E, A) is coherent if E is guarded, if it defines all variable of S, and if $Vars(E) = Vars(A)$.

Definition 12 (Soundness of gen). A function $gen(\_, \_)$ mapping each pair schema-assignment (S, A) to a set of JSON values is *sound* iff, for every coherent (S, E, A), if A is sound for E, then $gen(S, A) \subseteq \llbracket S \rrbracket_E$.

Definition 13 (Generativity of gen). A function $gen(\_, \_)$ mapping each pair schema-assignment to a set of JSON values is *generative* for a schema S iff for any E and A such that (S, E, A) is coherent:

(1) if $(\llbracket S \rrbracket_E \cap \mathcal{J}^1) \neq \emptyset$, then $gen(S, A) \neq \emptyset$;

(2) for any $i \geq 1$, if A is i-complete, and if $(\llbracket S \rrbracket_E \cap \mathcal{J}^{i+1}) \neq \emptyset$, then $gen(S, A) \neq \emptyset$.

gen is *generative for E* if it is generative for $E(y)$ for each variable $y \in Vars(E)$.

Intuitively, soundness of *gen* inductively implies that every assignment in every A^i is sound. Generativity implies that, if *gen* is used to compute A^{i+1} starting from an i-complete A^i, then A^{i+1} is $(i + 1)$-complete. Hence, if a variable x has a witness J of depth d, then $A^i(x) \neq \emptyset$ for every $i \geq d$.

We can now define our bottom-up algorithm (Algorithm 1) as follows:

Prepare(E) rewrites E and prepares all the extra variables needed for generation, as explained later. Then, we initialize $A[]$ as the minimal assignment A^0. We repeatedly execute a pass that sets $newA[y] = \mathtt{Gen}(E(y), A)$ for any y such that $A[y] == \emptyset$. If after that pass we have $newA[y] == A[y]$ for each y (no progress was made), then no further advance is possible, and the algorithm stops with "unsatisfiable". Otherwise, if we found a witness for the root variable x (that is, if $A[x] \neq \emptyset$) we stop the while loop and return the witness. At every pass through the loop, we either update $A[y]$

to a non-empty set for at least one variable that had $A[y] == \emptyset$ or we return "unsatisfiable", hence the algorithms traverses the loop at most once for every variable.

In the Supplementary Material, we prove that this algorithm is correct and complete.

Property 11 (Correctness 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 $A^i(x) \neq \emptyset$ and $A^i(x) \subseteq [[x]]_E$.
(2) If the algorithm terminates with "unsatisfiable", then $[[x]]_E = \emptyset$.
(3) The algorithm terminates after at most $|Vars(E)| + 1$ passes.

We can finally describe the phases of preparation and generation for all canonical groups, corresponding to the functions Prepare and Gen of Algorithm 1.

Preparation is a crucial phase, where the interactions between different object or array operators in a group are made explicit, and new variables are created in order to manage these interactions.

5.3 Object Group Preparation and Generation

Before describing object group preparation and generation, we introduce the notion of *constraints and requirements. Constraints* are object assertions that may be regarded as "universal quantifications" or "upper bounds" over the object members, while *requirements* are assertions that behave as "existential quantifiers" or "lower bounds".

More formally, an *object constraint* is an assertion $S = \text{props}(e : x)$ or $S = \text{pro}_0^M$ (notice the 0 minimum), and we observe that a *constraint* S has the following features:

(1) $\{\ \} \in [[S]]_E$
(2) $\{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E \Rightarrow \{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E$ — a constraint can be an upper bound on the object members, but not a lower bound.

An *object requirement* is an assertion $S = \text{pattReq}(e : x)$ or $S = \text{pro}_m^\infty$ with $m > 0$; a requirement has the following features:

(1) $\{\ \} \notin [[S]]_E$
(2) $\{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E \Rightarrow \{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E$ — a requirement can be a lower bound on the object members, but not an upper bound.

As a consequence, a possible algorithm to build an object is: start from the empty object, add one member at a time until all requirements are satisfied, and, for each member that is added, always verify that it satisfies *all* constraints.

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

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

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 $\texttt{"}a\texttt{"} : J$ where $J \in A^{i-1}(x)$, and a member $\texttt{"}c\texttt{"} : J'$, where J' is arbitrary.

Unfortunately, in the general case where we have overlapping 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 the following issues into account:

(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:

$$\{\ \mathsf{type}(\mathsf{Obj}), \mathsf{props}(p : x), \mathsf{pattReq}(e : y), \mathsf{pro}_1^1\ \}.$$

There are two distinct ways of producing a witness $\{\ k : J\ \}$ for the object above: *e-p*-matching, where we generate a k that matches $e \sqcap p$,[4] and a witness J for $y \wedge x$, and *e-non-p*-matching, where we generate a k that matches $e \sqcap \overline{p}$, and a witness J for y. This exemplifies the first two issues above:

(1) patterns: we need to compute which of the combinations $e \sqcap p$ and $e \sqcap \overline{p}$ have a non-empty language, in order to know which approaches are viable w.r.t. to pattern combination;
(2) new variables: to explore *e-p*-matching, we need a new variable whose body is $y \wedge x$, in order to generate a witness for this conjunctive schema, since the bottom-up algorithm generates witnesses for the variables, not for variable combinations.

Let us say that a member $k : J$ has shape $e : S$ when $k \in L(e)$ 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 $(e \sqcap p : y \wedge x)$ or $(e \sqcap \overline{p} : y)$.

To exemplify problem (3) — one member possibly satisfying many requirements — consider the following object group: $\{\ \mathsf{type}(\mathsf{Obj}), \mathsf{pattReq}(e_1 : y_1), \mathsf{pattReq}(e_2 : y_2), \mathsf{pro}_{min}^{Max}\}$.

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

(1) producing just one member with shape $e_1 \sqcap e_2 : y_1 \wedge y_2$;
(2) producing two members, with shapes $e_1 : y_1$ and $e_2 : y_2$.

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 equivalent to $y_1 \wedge y_2$.

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. The generative witness-generation function that is used during bottom-up evaluation, and that will be described in Section 5.3.3, uses this prepared form.

5.3.2 Object Group Preparation. Consider a generic object group

$$\{\ \mathsf{type}(\mathsf{Obj}),\ \mathsf{props}(p_1 : x_1), \ldots, \mathsf{props}(p_m : x_m),\ \mathsf{pattReq}(e_1 : y_1), \ldots, \mathsf{pattReq}(e_n : y_n),\ \mathsf{pro}_{min}^{Max}\ \}$$

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 members (k, J) where every member satisfies every constraint $\mathsf{props}(p_i : x_i)$ such that $k \in L(p_i)$, and such that every requirement $\mathsf{pattReq}(e_j : y_j)$ is satisfied by a member with $k \in L(e_j)$. Hence, every member in the witness 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') may match a member, 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:

[4] $e \sqcap p$ is the pattern that matches the intersection of $L(e)$ and $L(p)$, as defined in Section 3.5.

Definition 14 (Characteristic pattern). Given an object group $\{\mathsf{type}(\mathrm{Obj}), CP, RP, \mathrm{pro}_{min}^{Max}\}$ and two subsets $CP' \subseteq CP$ and $RP' \subseteq RP$, the characteristic pattern $cp(CP', RP')$ is defined as follows:

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

Consider for example the following object group, corresponding, modulo variable names, to a fragment of our running example (Figure 5(d)):

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

For space reason, we adopt the following abbreviations for the assertions in CP and RP:

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

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:

$cp(\{\},\{\})$	$=$	$\overline{b}\sqcap\overline{a}\sqcap\overline{a.*}$	$\equiv \overline{b}\sqcap\overline{a.*}$	$cp(\{pb\},\{\})$	$=$	$b\sqcap\overline{a}\sqcap\overline{a.*}$	$\equiv b$
$cp(\{\},\{ra\})$	$=$	$\overline{b}\sqcap a\sqcap\overline{a.*}$	$\equiv \emptyset$	$cp(\{pb\},\{ra\})$	$=$	$b\sqcap a\sqcap\overline{a.*}$	$\equiv \emptyset$
$cp(\{\},\{ras\})$	$=$	$\overline{b}\sqcap\overline{a}\sqcap a.*$	$\equiv a.+$	$cp(\{pb\},\{ras\})$	$=$	$b\sqcap\overline{a}\sqcap a.*$	$\equiv \emptyset$
$cp(\{\},\{ra,ras\})$	$=$	$\overline{b}\sqcap a\sqcap a.*$	$\equiv a$	$cp(\{pb\},\{ra,ras\})$	$=$	$b\sqcap a\sqcap a.*$	$\equiv \emptyset$

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 $\mathsf{props}(p_i : x_i) \in CP'$, since each relevant constraint must be satisfied, but, as far as the assertions $\mathsf{pattReq}(e_j : y_j) \in RP'$ 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 pro_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 pro_m^∞ requirement. The only choices that may describe a member are those where the set of strings $L(cp(CP', RP'))$ is not empty; we call them *non-cp-empty choices*.

Definition 15 (Choice, R-Choice, cp-empty choice). Given an object group $S = \{\, \mathsf{type}(\mathrm{Obj}), CP, RP, \mathrm{pro}_m^M \,\}$ with $CP = \{\mathsf{props}(p_i : x_i) \mid i \in 1..m\}$ and $RP = \{\mathsf{pattReq}(e_j : y_j) \mid j \in 1..n\}$, a choice for S, or just "a choice" when S is clear, 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:

$$s(CP', RP', RP'') = \bigwedge_{\text{props}(p:x)\in CP'} x \wedge \bigwedge_{\text{pattReq}(e:y)\in RP''} y.$$

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.

$s(\{\}, \{\}, \{\})$	$= x_t$	non-R-choice	$s(\{\}, \{ra, ras\}, \{\})$	$= x_t$	non-R-choice
$s(\{pb\}, \{\}, \{\})$	$= x$	non-R-choice	$s(\{\}, \{ra, ras\}, \{ra\})$	$= y1$	R-choice
$s(\{\}, \{ras\}, \{\})$	$= x_t$	non-R-choice	$s(\{\}, \{ra, ras\}, \{ras\})$	$= y2$	R-choice
$s(\{\}, \{ras\}, \{ras\})$	$= y2$	R-choice	$s(\{\}, \{ra, ras\}, \{ra, ras\})$	$= y1 \wedge y2$	R-choice

The schema of a choice is always a conjunction of variables, say $x_1 \wedge \ldots \wedge x_n$. During bottom-up generation, we generate witnesses for single variables, not for conjunctions, hence, we create a new variable y for each non-trivial conjunction $x_1 \wedge \ldots \wedge x_{n+1}$ (such as $y1 \wedge y2$ in the example), then we execute DNF normalization and canonicalization over $x_1 \wedge \ldots \wedge x_{n+1}$, transforming it into a canonical DNF S, then we add $y : S$ to the current environment, and we finally apply *preparation* to all the groups in S. In the example above, this may be the case for $y1 \wedge y2$.

This prepare-normalize-prepare loop can generate a huge number of variables; we keep their number under control using the ROBDDTab data structure introduced in Section 4.1, which allows us to create a new variable only when no existing variables is Boolean-equivalent to its body; this optimization also ensures that this phase can never generate an infinite loop (Property 12).

Hence, object preparation proceeds as follows:

(1) determine the set of non-cp-empty pairs (CP', RP');
(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 $x : vi$ to the environment, reapply DNF reduction to vi, apply preparation to the DNF-reduced conjunction.

Hereafter we use $var(C)$ to indicate either the variable x that is defined in step (2) above when $s(C)$ is an intersection, or $s(C)$ itself when $s(C)$ is already a variable.

Definition 16 ($var(C)$). For any non-cp-empty choice C that has been prepared, we use $var(C)$ to indicate the variable (equivalent to $s(C)$) that is associated to C after preparation.

When we describe object generation, we will show how the set of all prepared choices can be used in order to enumerate all possible ways of generating a witness for an object group.

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), at most n R-choices (one for each requirement), and at most $n + 1$ non-R-choices (one for each non-cp-empty pair).

The cost of this phase is in $O(2^{poly(N)})$, and the use of ROBDD is crucial for this limit. Our experiments show that this cost is, for most real-world schemas, tolerable.

Property 12. Object preparation can be performed in $O(2^{poly(N)})$ time.

Proof. Since before preparation we have at most $O(N)$ distinct variables (where N is the input size), step (2) may generate at most $O(2^N)$ variable conjunctions that are not ROBDD equivalent to

any other conjunction, each of which has a body which can be prepared in time $O(2^{poly(N)})$, so that the global cost is in $O(2^{poly(N)})$. ROBDD reduction is invoked at most $O(2^{poly(N)})$ times on variables whose Boolean body is in $O(N)$, since it is a conjunction of variables that were already present at the end of the preparation phase. □

5.3.3 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; this is the function Gen used in Algorithm 1.

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 member in the generated object, and vice versa; (2) we verify that the list is *pattern-viable*, i.e., that it does not require two members 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* lists of choices (defined below), and we prove that this subset 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:

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

then $\{\ \text{"}a\text{"} : 1, \text{"}b\text{"} : 1\ \}$ is described by each the following four choice lists (and by others), where each choice could be used to generate/describe each of the two members (all choices C in the four lists below have $cp(C) = \text{"}a|b\text{"}$, and they have $s(C)$ equal to either x_t, x, y, or $x \wedge y$):

$$\begin{array}{lll} CL_1 = & [\ (\{\}, \{rx, ry\}, \{rx\}), & (\{\}, \{rx, ry\}, \{ry\}) \quad] \\ CL_2 = & [\ (\{\}, \{rx, ry\}, \{rx, ry\}), & (\{\}, \{rx, ry\}, \{\}) \quad] \\ CL_3 = & [\ (\{\}, \{rx, ry\}, \{rx, ry\}), & (\{\}, \{rx, ry\}, \{rx, ry\}) \quad] \\ CL_4 = & [\ (\{\}, \{rx, ry\}, \{rx, ry\}), & (\{\}, \{rx, ry\}, \{rx\}) \quad] \end{array}$$

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 20.

Definition 17 (Disjoint solution, Minimal disjoint solution). For any object group

$$S = \{\text{type(Obj)}, CP, RP, \text{pro}_m^M\},$$

and any list $\mathbb{C} = [\ (C_1, R'_1, R''_1), \ldots, (C_n, R'_n, R''_n)\ ]$ of n choices for that group:

(1) $\mathbb{C}$ is a *solution* for RP if: $\bigcup_{i \in \{1..n\}} R''_i = RP$;
(2) $\mathbb{C}$ is *a disjoint solution* if it is a solution and if: $\forall 1 \le i < j \le n.\ R''_i \cap R''_j = \emptyset$;
(3) $\mathbb{C}$ is *a minimal solution* if it is a solution and every choice in $\mathbb{C}$ is an R-choice;
(4) $\mathbb{C}$ is a solution for S if it is a solution for RP and $m \le n \le M$.

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

A solution $\mathbb{C}$ for an object group S *describes* some objects (Definition 18); every object J described by a solution $\mathbb{C}$ for an object group S is a witness for S (Property 13).

Definition 18 (C describes-in-A a member $k : J$, $\mathbb{C}$ describes-in-A an object J). A choice $C = (CP', RP', RP'')$ for a prepared object group "*describes in an assignment A*" a member $k : J$, iff $k \in L(cp(C))$

and $J \in A(var(C))$. A choice list $\mathbb{C}$ *describes* in A an object J if there is a bijection mapping each member $k : J'$ in J to a choice C in $\mathbb{C}$ such that C describes $k : J'$.

Property 13 (J described by a solution $\Rightarrow$ J is a witness). For any prepared object group $S = \{ \mathsf{type(Obj)}, CP, RP, \mathsf{pro}_m^M \}$ with a corresponding environment E, if $\mathbb{C}$ is a solution for S, if J is described in A by $\mathbb{C}$ and A is sound for E, then $J \in [\![S]\!]_E$.

Object generation for a group $\{ \mathsf{type(Obj)}, CP, RP, \mathsf{pro}_m^M \}$ is based 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 prepared choice C is *Witnessed*, or *NonWitnessed*, when $var(C)$ is *Witnessed*, or is *NonWitnessed*. In order to generate a witness from A^i, we first generate a *disjoint minimal solution* $\mathbb{C}$ for RP with $|\mathbb{C}| \leq M$, only using R-choices that are *Witnessed* in A^i. Then, to deal with the uniqueness of property names in an object, 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 19. A set of choices $\mathbb{C}$ is *pattern-viable* iff for every pair (CP', RP'), the number of choices in $\mathbb{C}$ that match $(CP', RP', \_)$ 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:

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

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

Finally, for each viable disjoint solution, in case it does not satisfy pro_m^∞, indicated by $missing > 0$ (line 5), we try and extend the solution by adding some *Witnessed* non-R-choices that produce a viable solution (lines 5-10). Observe that a 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; hence, in line 6 we can pick the same NRC many times, unless at some point it is moved in those which make the combination *[NRC]++Solution* no more viable. If the current solution is not long enough and has no viable extension with $missing = 0$, we need to start from a different minimal disjoint solution. If no viable minimal disjoint solution admits a viable extension of length at least m (according to the current assignment), then the algorithm returns "NonWitnessed". Otherwise, we use the extended solution $\mathbb{C}'$ to build a witness (WitnessFrom(Solution)): 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 — this is the n-enumeration problem for EEREs that we introduced in Section 3.5.

Theorem 20 (Soundness and generativity). *Algorithm* Gen *is sound and generative.*

Proof. Our algorithm is sound by construction. For generativity, assume that the object group

$$S = \left\{ \mathsf{type(Obj)}, CP, RP, \mathsf{pro}_{min}^{Max} \right\},$$

has a witness of depth $d + 1$ in $[\![S]\!]_E$. Assume that A is d-complete for E. We want to prove that Gen, applied to S and A, will generate at least one witness (whose depth is not necessarily $d + 1$). Let $J = \{a_1 : J_1, \ldots, a_l : J_l\}$ be a witness for S in E with depth $d + 1$. We can now extract from $\{a_1 : J_1, \ldots, a_l : J_l\}$ a set of choices $\{(C_i', R_i', R_i'')\}$ with $i \in \{1..l\}$, as follows. C_i' and R_i' are defined

ALGORITHM 2: Object witness generation

```
1  Gen(RPart, WitnessedRChoices,WitnessedNRChoices, min, Max)
2    for Solution in minDisjointSols (WitnessedRChoices,RPart,Max) where (viable(Solution)) do
3      missing := min − size(Solution);
4      nonViableNRChoices := ∅;
5      while (missing > 0 and nonViableNRChoices !=WitnessedNRChoices) do
6        NRC ← chooseFrom(WitnessedNRChoices-nonViableNRChoices);
7        if (viable([NRC]++Solution)) then
8          Solution := [NRC]++Solution;
9          missing := missing-1;
10       else nonViableNRChoices := [NRC]++nonViableNRChoices;
11     if (missing == 0) then return ("Witnessed", WitnessFrom(Solution)) ;
       // if we arrive here, then missing != 0, hence we move to the next Solution
     // if we arrive here, then no viable and completable solution has been found
12   return ("NonWitnessed");
```

as the only pair (C'_i, R'_i) whose language includes a_i. In order to define R''_i, we observe that, since J satisfies RP, then, we can associate to each S in RP one member i such that $a_i : J_i$ satisfies S — if many such members exist, we just choose one. The inverse of this function associates to each member i a subset R''_i of R'_i. The collection of choices $\mathbb{C} = \{ (C'_i, R'_i, R''_i) \mid i \in \{1..l\} \}$ that we have defined is actually a multiset, since a non-R-choice may appear more than once, and is a disjoint solution since, by construction, $\bigcup_{i\in\{1..l\}} R''_i = RP$, $l \leq Max$, and $1 \leq i < j \leq l \implies R''_i \cap R''_j = \emptyset$, since every requirement is mapped to exactly one member. We now prove that all these choices are *Witnessed* in A. To this aim, consider a choice $C = (C'_i, R'_i, R''_i)$ in $\mathbb{C}$ and the member $a_i : J_i$ that we used to define it. By construction, the schema $s(C)$ is the conjunction of the variables of the constraints C'_i and the variables of the requirements R''_i. J_i is in $[\![x]\!]_E$ for any x that comes from C'_i, because J is a witness for S and C'_i is defined as the set of all constraints of S that must be satisfied by a member whose name is a_i. The set R''_i is, by construction, a set of requirement that are satisfied by $a_i : J_i$, hence $J_i \in [\![x]\!]_E$ for any x that comes from a requirement in R''_i. Hence, $J_i \in [\![x]\!]_E$ for all the variables x in $s(C'_i, R'_i, R''_i)$, hence $J_i \in [\![s(C)]\!]_E$, hence, by definition of $var(C)$, $J_i \in [\![var(C)]\!]_E$. Since J has depth $d + 1$, then $\delta(J_i) \leq d$, hence $J_i \in [\![var(C)]\!]_E \implies A(var(C)) \neq \emptyset$ since A is d-complete, that is, the choice C is *Witnessed* in A.

Now we prove that the existence of this solution $\mathbb{C}$ witnessed in A implies that our algorithm generates a witness. To this end, we remove every non-R-choice from $\mathbb{C}$, and so we get a collection $\mathbb{C}'$ that is a minimal disjoint solution. If $min > |\mathbb{C}'|$ (where min comes from the pro^{Max}_{min} assertion of S), then we choose $min - |\mathbb{C}'|$ non-R-choices out of $\mathbb{C}$ and add them to $\mathbb{C}'$. Being a sub-multiset of $\mathbb{C}$, the result is viable and, by construction, it is an extension of a minimal disjoint solution $\mathbb{C}'$ with a multiset of $min - |\mathbb{C}'|$ non-R-choices. Our algorithm scans every such extension of every minimal disjoint solution, hence, if it is not stopped before because it finds a different solution, it finds this one; in both cases, the algorithm generates a witness. □

Property 14 (Complexity). Given a schema of size N, each run of the `Gen` algorithm has a time complexity in $O(2^{poly(N)})$.

Proof. Let N we the size of the original schema. For any object group, RP has at most N elements, and any choice has a size that is in $O(N)$. Let M be an upper bound for the number of non-cp-empty choices for an arbitrary object group. Since every minimal disjoint solution contains at most $|RP| \leq N$ choices, we can generate all minimal disjoint solutions by scanning the list of all N-tuples

of non-cp-empty choices, which can be done in time $O(M^N)$. For each minimal solution, we need to scan the list of all non-R-choices for at most *min* times, which adds another $O(M^{min})$ factor; since $min \leq N$ by the small constants assumption, we arrive at $O(M^{2N})$ lists of choices to scan. For every list $\mathbb{C}$, in order to verify its viability and to generate the witness when a witness exists, we need to solve at most $|\mathbb{C}|$ times the i-enumeration problem, once for each group of choices with the same pattern $cp(C)$, each time with $i \leq |\mathbb{C}|$, since we have at most $|\mathbb{C}|$ choices with the same pattern. The pattern expression $cp(C)$ of each choice $C \in \mathbb{C}$ has a size that is in $O(poly(N))$, and $|\mathbb{C}|$ is smaller than N, hence running $|\mathbb{C}|$ times the i-enumeration problem has a cost that is $O(2^{poly(N)})$, hence we can examine the $O(M^{2N})$ solutions in time $O(M^{2N} \cdot 2^{poly(N)})$ (we ignore the polynomial factors). Since M is in $O(2^{poly(N)})$, then $O(M^{2N}) = O(2^{poly(N)})$, hence each pass of object generation is in $O(2^{poly(N)})$ for each prepared object group. Since we have less than $O(2^{poly(N)})$ groups, each pass of object generation is in $O(2^{poly(N)} \times 2^{poly(N)}) = O(2^{poly(N)})$. □

5.4 Array Group Preparation and Generation

As with objects, before describing array preparation and generation, we must introduce a distinction between constraints and requirements: we say that an assertion $S = \mathsf{contAfter}(i^+ : x)$ or $S = \mathsf{cont}_i^\infty(x)$ with $i > 0$ is a *requirement*, since it is not satisfied by [ ] and, if J^+ extends J, then $J \in [\![S]\!]_E \Rightarrow J^+ \in [\![S]\!]_E$; requirements are *lower bounds* for the list of the array items.

We say that an assertion $S = \mathsf{item}(l : x)$, $S = \mathsf{items}(i^+ : x)$, or $S = \mathsf{cont}_0^j(x)$ is a *constraint*, since it is satisfied by [ ] and if J^+ extends J, then $J^+ \in [\![S]\!]_E \Rightarrow J \in [\![S]\!]_E$; constraints are *upper bounds* for the list of the array items.

An assertion $S = \mathsf{cont}_i^j(x)$ with $i \neq 0$ and $j \neq \infty$ combines a requirement and a constraint.

5.4.1 Array Group Preparation. An array group is a set of assertions with the following shape:

$$\{\ \mathsf{type}(\mathsf{Arr}), IP, AP, KP\ \}.$$

Here, IP is a set of *item* constraints $\mathsf{item}(l : x)$ and $\mathsf{items}(i^+ : x)$, AP is a set of *contains-After* requirements with shape $\mathsf{contAfter}(i^+ : x)$, KP is a set of *counting* assertions $\mathsf{cont}_i^j(x)$, where every assertions combines a requirement $\mathsf{cont}_i^\infty(x)$ and a constraint $\mathsf{cont}_0^j(x)$.[5]

Arrays and objects look very similar, since they are both finite mappings from labels to values, but arrays have some extra issues:

(1) Arrays have a domain downward closure constraint, which specifies that, for every label $n \geq 1$, when a value is associated to a label $n + 1$, then some value is associated to n as well; objects do not have anything similar.
(2) The $\mathsf{cont}_{min}^\infty(x)$ requirement depends on counting, while $\mathsf{pattReq}(a : x)$ only specifies the existence of at least one member matching a with schema x, with no counting ability.
(3) The $\mathsf{cont}_0^{Max}(x)$ constraint specifies an upper bound that depends on the variable x, which forces us to count not only how many items satisfy x but, crucially, how many items do *not* satisfy x.

Another fundamental difference is that, while the arbitrary intersections of n patterns may describe 2^n different languages, the arbitrary intersections of n different intervals $[0, i]$ describe exactly the same n intervals; this implies that the best way to explore the space of solutions is potentially very different for objects and arrays.

[5]For the sake of simplicity, in our formal treatment we do not distinguish $\mathsf{cont}_i^j(x_\mathbf{t})$ from the other counting assertions, where $x_\mathbf{t}$ here indicates the variable whose body is $\mathbf{t}$, although in the implementation we actually exploit its special properties for efficiency reasons.

Example 2. Consider for example the following array group.

$$\{\, \mathsf{type}(\mathsf{Arr}), \mathsf{item}(2 : x), \mathsf{contAfter}(0^+ : y), \mathsf{cont}^1_1(z), \mathsf{cont}^2_2(x_\mathsf{t}) \}.$$

It describes an array of exactly two elements, because of $\mathsf{cont}^2_2(x_\mathsf{t})$. The one at position 2 must satisfy x. At least one of the two elements must satisfy y. One, but only one, of the two elements must satisfy z, because of $\mathsf{cont}^1_1(z)$.

Let us say that an array has shape $[S_1, \ldots, S_k]$ if it contains exactly k items $[J_1, \ldots, J_k]$, and if each item J_i satisfies S_i. Then, the group above is satisfied by arrays with one of the following four shapes:

$$[\ y \wedge z,\ x \wedge co(z)\ ], \quad [\ y \wedge co(z),\ x \wedge z\ ],\ [\ z,\ x \wedge y \wedge co(z)\ ], \quad [\ co(z),\ x \wedge y \wedge z\ ].$$

We recognize the two problems seen with objects: interaction between constraints and requirements, resulting in conjunctions of x with other variables in position 2, and the possibility of one element to satisfy two requirements, resulting in $y \wedge z$ conjunctions, but we have the extra problem of the upper bound, that results in the presence of the dual variable $co(z)$ in some positions.

Hence, our algorithm to prepare arrays and to generate the corresponding witnesses is quite different from that of objects, although we try and keep a similar flavor, to make it easier to understand. It obviously differs in the presence of dual variables like $co(z)$, motivated by upper bounds, but it also differs in the strategy that we use to explore the space of witnesses. Instead of starting the exploration from the requirements (the non-R-choices), here we are guided by the domain closure constraint, hence we start the exploration from the first position of the array.

We first define a notion of head-length for an array group S (Definition 21). For example, for an array group $\{\mathsf{item}(1 : S_1), \mathsf{items}(3^+ : S_2)\}$ we say that its head-length is 3 ($\max(1, 3)$), and we say that, for any witness $J = [J_1, \ldots, J_n]$, the elements (if any) in positions from 1 to 3 form the *head* and those from positions 4 to n (if they exist) form the *tail*. The essential property of the tail is that the elements inside the tail of the witness J may be freely permuted, and the result is still guaranteed to be a witness, while the actual position of the elements in the head may be relevant. The head-length n may be 0, and actually, this is the most common head-length that we encounter in practice. The interval of an assertion $In(S)$ is, intuitively, the interval of positions of the array that are "relevant", that is, where the assertion puts some constraints, or where the requirements may be satisfied; this interval may be included in the head of the group, or in the tail, or it may cross both. Head-length of the group and interval of each assertion are defined as follows:

Definition 21 ($[i, j]$, $HL(S)$, $In(S)$). $[i, j]$, with $i \in \mathbb{N}$, $j \in \mathbb{N}^\infty$, denotes the interval between i and j, which is infinite when $j = \infty$, and is empty when $i > j$. The head-length $HL(S)$ and the interval $In(S)$ of an array ITO S, and of an array group, are defined as follows, where $\max(\emptyset) = 0$:

$l \in \mathbb{N}_{>0},\ i \in \mathbb{N},\ j \in \mathbb{N}^\infty,\ [i, j] =_{def} \{\, k \mid k \in \mathbb{N},\ i \le k \le j \,\}$:

$HL(\mathsf{item}(l : S))$	$=$	l	$HL(\mathsf{items}(i^+ : S))$	$=$	i
$HL(\mathsf{contAfter}(i^+ : S))$	$=$	i	$HL(\mathsf{cont}^j_i(S))$	$=$	0
$HL((\{\mathsf{type}(\mathsf{Arr}), IP, AP, KP\})$	$=$	$\max_{S \in IP \cup AP}(HL(S))$			
$In(\mathsf{item}(l : S))$	$=$	$[l, l]$	$In(\mathsf{items}(i^+ : S))$	$=$	$[i + 1, \infty]$
$In(\mathsf{contAfter}(i^+ : S))$	$=$	$[i + 1, \infty]$	$In(\mathsf{cont}^j_i(S))$	$=$	$[1, \infty]$

Property 15 (Free permutability of tail). If S is an array group, $J = [J_1, \ldots, J_n] \in [\![S]\!]_E$, for all i, j with $HL(S) < i \le j \le n$, if J' is obtained from J by exchanging J_i with J_j, then $J' \in [\![S]\!]_E$.

Property 1 (Irrelevance out of interval). If S is an array assertion, $J = [J_1, \ldots, J_n] \in [\![S]\!]_E$, if J' is an array that coincides with J on all positions in $In(S)$, then $J' \in [\![S]\!]_E$.

In order to define a choice, we need a last definition: for a set of assertions $\mathcal{S}$, we define its restriction to $[i, j]$, denoted by $\mathcal{S} \cap [i, j]$, as the subset of $\mathcal{S}$ containing the assertions whose interval intersects $[i, j]$.

Definition 22 ($\mathcal{S} \cap [i, j]$). $\mathcal{S} \cap [i, j] = \{ S \mid S \in \mathcal{S},\ ([i, j] \cap In(S)) \neq \emptyset \}$.

We can now define the notion of a choice for array preparation and generation.

Definition 23 (Choice for an array group). Given an array group $\{ \text{type(Arr)}, IP, AP, KP \}$ with $h = HL(IP \cup AP)$, a choice for the array group is a quintuple $([i, j], IP', AP', KP^+, KP^-)$ where:

(1) $[i, j]$ is either $[i, i]$ with $i \leq h$ (head singleton) or $[h + 1, \infty]$ (generic tail position);
(2) IP' is equal to $IP \cap [i, j]$;
(3) AP' is a subset of $AP \cap [i, j]$;
(4) KP^+ is a subset of KP;
(5) KP^- is a subset of $KP \setminus KP^+$.

So, a choice describes an element in a precise position i in the head ($[i, i]$), or an element somewhere in the tail ($[h + 1, \infty]$). Observe that, for a given interval interval $[i, j]$, the IP' constraint component is fixed, while we have many alternatives for AP', KP^+ and KP^-. Differently from object choices, in array choices the label space is not described by a pair of sets of assertions (CP', RP'), but just by an interval $[i, j]$, while the schema is a bit more complex since it has three positive components IP', AP', and KP^+, playing the roles of CP' and RP'', but also a negative component KP^-. Observe that, while IP' and AP' are restricted to the assertions that apply to $[i, j]$, we do not have this restriction for KP, since every counting assertion analyzes all positions of the array. Hence, the schema of a choice is defined as follows:

Definition 24 ($s([i, j], IP', AP', KP^+, KP^-)$).

$$s([i, j], IP', AP', KP^+, KP^-) = \quad (\bigwedge_{(\text{item}(l:x)) \in IP'} x) \wedge (\bigwedge_{(\text{items}(i^+:x)) \in IP'} x) \wedge (\bigwedge_{(\text{contAfter}(i^+:x)) \in AP'} x) \wedge (\bigwedge_{(\text{cont}_i^j(x)) \in KP^+} x) \wedge (\bigwedge_{(\text{cont}_i^j(x)) \in KP^-} co(x)).$$

As with object groups, we do not really need to generate all possible choices, and different strategies are possible to reduce the search space. In our implementation, we limit ourselves to the choices where $KP^- = KP \setminus KP^+$, which we call here the co-maximal choices. We prove later that this strategy ensures the generativity property that we need. More optimized strategies would be possible, but we believe that they are not worth the effort, since in practice the array groups that we have to deal with are usually quite simple.

Example 3. Consider again the array group $S = \{ \text{type(Arr)}, IP, AP, KP \}$ of Example 2, where $IP = \{\text{item}(2 : x)\}$, $AP = \{\text{contAfter}(0^+ : y)\}$ and $KP = K_1 \cup K_2$, with $K_1 = \{\text{cont}_1^1(z)\}$ and $K_2 = \{\text{cont}_2^2(x_\mathfrak{t})\}$. Head-length for this schema is 2. The table below shows the schemas that correspond to the 24 co-maximal choices for S. Observe that j in $[i, j]$ is determined by i, the assertion set IP' is determined by $[i, j]$, and also KP^- is determined by KP^+, since the choices are co-maximal.

All the schemas that end with $x_\mathfrak{f}$ are actually rewritten, by ROBDD reduction, as just $x_\mathfrak{f}$.

All the shapes listed in Example 2 are composed by pairing the schema of a choice for $[1, 1]$ and that of a choice for $[2, 2]$. For example, the shape $[y \wedge z, x \wedge co(z)]$ corresponds to a choice list $[([1, 1], \emptyset, AP, KP, \emptyset), ([2, 2], IP, \emptyset, K_2, K_1)]$, and similarly for the other four shapes. Not every pair of choices describes an instance of S; when a sequence of choices $\mathbb{C}$ guarantees the satisfaction of all requirements and all constrains of its array group, we say that $\mathbb{C}$ is a solution. In the next section, we will describe a generation algorithm that builds solutions.

Our array generation technique relies on array preparation, which consists of the following steps, to be executed for every array group $S = \{\, \mathsf{type(Arr)}, IP, AP, KP \,\}$.

(1) compute $h = HL(IP, AP)$;
(2) for each interval $[i, i]$ corresponding to an $i \in [1, h]$, and for each subset AP' of AP and KP' of KP, produce the corresponding co-maximal choice:

$$C = ([i, i], IP \cap [i, i], AP', KP', KP \setminus KP'),$$

and check whether the variable intersection $s(C)$ (Definition 24) is ROBDD-equivalent to some existing variable y_e, and, if not, create a new variable y_n with body $s(C)$, and apply preparation to the body $s(C)$ of this new variable; as in the case of object preparation, define $var(C)$ to be either y_e or y_n, according to whether y_e existed or not;
(3) do the same for the interval $[h + 1, \infty]$, and for each subset AP' of AP and KP' of KP.

As happens with object preparation, also array preparation has an exponential cost that is quite low in practice, since in the vast majority of cases the head-length of array groups is zero or one, and the set $AP \cup KP$ is either empty or a singleton. For this reason, we did not put any special effort into the optimization of this phase.

Property 16. Array preparation can be performed in time $O(2^{poly(N)})$, where N is the size of the input schema.

5.4.2 Witness Generation from a Prepared Array Group. Array preparation applied to an array group $\{\, \mathsf{type(Arr)}, IP, AP, KP \,\}$ with head-length h produces a set of co-maximal choices, each fully determined by an interval $[i, j]$ with shape $[i, i]$ when $i \leq h$, or $[h + 1, \infty]$ otherwise, and by two subsets AP', KP' of AP, KP. We indicate with $([i, .], ., AP', KP', .)$ the only co-maximal choice that is determined by i, AP', KP', as follows:

$$\begin{array}{rrcl} 1 \leq i \leq h: & ([i, .], ., AP', KP', .) & = & ([i, i], IP \cap [i, i], AP', KP', KP \setminus KP') \\ i = h + 1: & ([i, .], ., AP', KP', .) & = & ([i, \infty], IP \cap [i, \infty], AP', KP', KP \setminus KP'). \end{array}$$

A choice $([i, .], ., AP', KP', .)$ is a *head choice* when $i \leq h$, and is a *tail choice* when $i = h + 1$. As with objects, at any pass j of the generation algorithm, a choice C is *Witnessed* or *NonWitnessed*, depending on whether $A^j(var(C))$ is empty or not.

Given a list of choices $\mathbb{C}$ and an assertion S, we define the *incidence* of $\mathbb{C}$ over S ($I_{\mathbb{C}}(S)$) to be the number of elements of $\mathbb{C}$ that contain S in their AP' or KP' component, so that, referring to Example 3, if $\mathbb{C} = [([1, .], ., AP, \{K_1, K_2\}, .), ([2, .], ., \emptyset, \{K_2\}, .)]$, then $I_{\mathbb{C}}(K_1) = 1$ and $I_{\mathbb{C}}(K_2) = 2$.

Definition 25 (Incidence of $\mathbb{C}$ over S — $I_{\mathbb{C}}(S)$).

$$I_{\mathbb{C}}(S) \quad = \quad |\{\, ([i, .], ., AP', KP', .) \mid ([i, .], ., AP', KP', .) \in \mathbb{C},\ S \in AP' \vee S \in KP' \,\}|.$$

We define now a well-formed list of choices to be a list that describes a set of consecutive positions, which may be empty. For example, with head length 4, the following three lists are well-formed: [ ], [([2, 2], . . .), ([3, 3], . . .)], [([3, 3], . . .), ([4, 4], . . .), ([5, ∞], . . .), ([5, ∞], . . .)]; to simplify induction proofs, we do not here force a well-formed list to start from position 1.

Definition 26 (Well-formed for h). A list of array choices is well-formed for head-length h iff

(1) every choice in the list has either an interval $[i, i]$ with $i \leq h$ or the interval $[h + 1, \infty]$;
(2) if two consecutive choices in the list have intervals $[i, .]$ and $[j, .]$, then either $j = i + 1$ or $j = i = h + 1$.

We say that a well-formed list of choices $\mathbb{C}$ that starts from the first position is a solution for $AP \cup KP$ when the incidence of $\mathbb{C}$ on the assertions of $AP \cup KP$ satisfies all requirements of AP and KP and does not violate any upper bound constraint of KP, as follows:

Table 2. Choices for $S = \{\, \mathsf{type(Arr)}, \mathsf{item}(2:x), \mathsf{contAfter}(0^+:y), \mathsf{cont}_1^1(z), \mathsf{cont}_2^2(x_\mathbf{t})\,\}$, where $IP = \{\mathsf{item}(2:x)\}$, $AP = \{\mathsf{contAfter}(0^+:y)\}$, $K_1 = \{\mathsf{cont}_1^1(z)\}$, $K_2 = \{\mathsf{cont}_2^2(x_\mathbf{t})\}$, and $KP = K_1 \cup K_2$

$[\mathbf{i}, j]$	IP'	$\mathbf{AP'}$	$\mathbf{KP^+}$	KP^-	$s(C)$	$[\mathbf{i}, j]$	IP'	$\mathbf{AP'}$	$\mathbf{KP^+}$	KP^-	$s(C)$
$[\mathbf{1},1]$	$\emptyset$	AP	KP	$\emptyset$	$y \wedge z \wedge x_\mathbf{t}$	$[\mathbf{2},2]$	IP	$\emptyset$	KP	$\emptyset$	$x \wedge z \wedge x_\mathbf{t}$
$[\mathbf{1},1]$	$\emptyset$	AP	K_1	K_2	$y \wedge z \wedge x_\mathbf{f}$	$[\mathbf{2},2]$	IP	$\emptyset$	K_1	K_2	$x \wedge z \wedge x_\mathbf{f}$
$[\mathbf{1},1]$	$\emptyset$	AP	K_2	K_1	$y \wedge co(z) \wedge x_\mathbf{t}$	$[\mathbf{2},2]$	IP	$\emptyset$	K_2	K_1	$x \wedge co(z) \wedge x_\mathbf{t}$
$[\mathbf{1},1]$	$\emptyset$	AP	$\emptyset$	KP	$y \wedge co(z) \wedge x_\mathbf{f}$	$[\mathbf{2},2]$	IP	$\emptyset$	$\emptyset$	KP	$x \wedge co(z) \wedge x_\mathbf{f}$
$[\mathbf{1},1]$	$\emptyset$	$\emptyset$	KP	$\emptyset$	$z \wedge x_\mathbf{t}$	$[\mathbf{3},\infty]$	$\emptyset$	AP	KP	$\emptyset$	$y \wedge z \wedge x_\mathbf{t}$
$[\mathbf{1},1]$	$\emptyset$	$\emptyset$	K_1	K_2	$z \wedge x_\mathbf{f}$	$[\mathbf{3},\infty]$	$\emptyset$	AP	K_1	K_2	$y \wedge z \wedge x_\mathbf{f}$
$[\mathbf{1},1]$	$\emptyset$	$\emptyset$	K_2	K_1	$co(z) \wedge x_\mathbf{t}$	$[\mathbf{3},\infty]$	$\emptyset$	AP	K_2	K_1	$y \wedge co(z) \wedge x_\mathbf{t}$
$[\mathbf{1},1]$	$\emptyset$	$\emptyset$	$\emptyset$	KP	$co(z) \wedge x_\mathbf{f}$	$[\mathbf{3},\infty]$	$\emptyset$	AP	$\emptyset$	KP	$y \wedge co(z) \wedge x_\mathbf{f}$
$[\mathbf{2},2]$	IP	AP	KP	$\emptyset$	$x \wedge y \wedge z \wedge x_\mathbf{t}$	$[\mathbf{3},\infty]$	$\emptyset$	$\emptyset$	KP	$\emptyset$	$z \wedge x_\mathbf{t}$
$[\mathbf{2},2]$	IP	AP	K_1	K_2	$x \wedge y \wedge z \wedge x_\mathbf{f}$	$[\mathbf{3},\infty]$	$\emptyset$	$\emptyset$	K_1	K_2	$z \wedge x_\mathbf{f}$
$[\mathbf{2},2]$	IP	AP	K_2	K_1	$x \wedge y \wedge co(z) \wedge x_\mathbf{t}$	$[\mathbf{3},\infty]$	$\emptyset$	$\emptyset$	K_2	K_1	$co(z) \wedge x_\mathbf{t}$
$[\mathbf{2},2]$	IP	AP	$\emptyset$	KP	$x \wedge y \wedge co(z) \wedge x_\mathbf{f}$	$[\mathbf{3},\infty]$	$\emptyset$	$\emptyset$	$\emptyset$	KP	$co(z) \wedge x_\mathbf{f}$

Definition 27 (Solution for array groups). Fixed an array group $\{\, \mathsf{type(Arr)}, IP, AP, KP \,\}$ with head-length h, a co-maximal choice list $\mathbb{C}$ is a solution for the array group iff:

(1) it is well-formed for h;
(2) either $\mathbb{C}$ is empty or the first choice in $\mathbb{C}$ has interval $[1, \_]$;
(3) for every assertion $\mathsf{cont}_m^M(x) \in KP$ we have $I_\mathbb{C}(\mathsf{cont}_m^M(x)) \leq M$;
(4) for every assertion $\mathsf{cont}_m^M(x) \in KP$ we have $I_\mathbb{C}(\mathsf{cont}_m^M(x)) \geq m$;
(5) for every requirement $S \in AP$ we have $I_\mathbb{C}(S) > 0$.

Example 4. Going back to the array schema of Examples 2 and 3, and Table 2, we observe that, for example, the choice list $[([1,.],.,AP,KP,.),([2,.],.,\emptyset,K_2,.)]$ is a solution, since every assertion has a correct incidence: (a) $I_\mathbb{C}(AP) = I_\mathbb{C}(\mathsf{contAfter}(0^+:y)) \geq 1$, (b) $I_\mathbb{C}(K_1) = I_\mathbb{C}(\mathsf{cont}_1^1(z)) = 1$, (c) $I_\mathbb{C}(K_2) = I_\mathbb{C}(\mathsf{cont}_2^2(x_\mathbf{t})) = 2$. The schemas of $([1,.],.,AP,KP,.)$ and $([2,.],.,\emptyset,K_2,.)$ are $y \wedge z \wedge x_\mathbf{t}$ and $x \wedge co(z) \wedge x_\mathbf{t}$ (Table 2), hence this solution corresponds to the shape $[y \wedge z, x \wedge co(z)]$ of Example 2. A choice list $\mathbb{C} = [([1,.],.,AP,KP,.),([2,.],.,\emptyset,K_2,.),([3,.],.,\emptyset,K_2,.))]$ is not a solution since $I_\mathbb{C}(K_2) = I_\mathbb{C}(\mathsf{cont}_2^2(x_\mathbf{t})) = 3$ violates the upper bound.

A list of array choices describes an array, in a given assignment A, in the same way as a list of object choices describes an object:

Definition 28 ($\mathbb{C}$ describes-in-A array J). A choice $C = ([i,j],\ldots)$ for a prepared array group *describes in an assignment* A an element J_l of an array $[J_1,\ldots,J_n]$, iff $l \in [i,j]$ and $J_l \in A(var(C))$. A choice list $[C_1,\ldots,C_n]$ *describes* in A an array $J = [J_1,\ldots,J_n]$ if every C_l describes in A the element J_l.

When a *solution* of S *describes* in a sound assignment A an array J, then J is a witness for S:

Property 17. For any prepared array group $S = \{\, \mathsf{type(Arr)}, IP, AP, KP \,\}$ with the corresponding environment E and choices $\mathbb{C}$, if (a) A is sound for E, if (b) the choice list $\mathbb{C}'$ over $\mathbb{C}$ is a co-maximal solution for S, and if (c) J is described in A by $\mathbb{C}$, then: $J \in [\![S]\!]_E$.

We finally need a notion of *useful choices*, which is similar in spirit to the *R-choices* that we defined for the object case, and which will be crucial to ensure the termination of the algorithm: a choice C is *useful* for a set $AP \cup KP$ iff some assertion in $AP \cup KP$ is affected by C.

ALGORITHM 3: Pseudo-code for array solution generation

```
// Given an assertion list aList, we assume the existence of a choice-list pref with length
   prefLen, and where prefInc[Assertion] is the incidence of pref on Assertion, and we return
   a list of choice-lists that can be appended to pref in order to get a solution for aList
cList(hLen, aList, wChoices, prefLen, prefInc)
    if everyReqSatisfied(aList,prefInc) then return [ ];
    uWChoices ← usefulChoices(hLen, wChoices, prefLen, prefInc);
    for C in uWChoices where inInterval(prefLen+1,C) do
        newPrefInc ← updateIncAfterChoice(prefInc, C);
        if maxViolated(aList, newPrefInc) then continue;
        else
            restSolution = cList(hLen, aList, uWChoices, prefLen+1, prefLen, newPrefInc);
            if restSolution is not "NoSolution" then return ([C] ++ restSolution);
    return "NoSolution";
// return the choices that are still useful after a prefix described by prefLen and prefInc
usefulChoices(hLen, choices, prefLen, prefInc)
    useful = [ ];
    // Every head choice is "useful", if the start position of its interval is > prefLen
    for C in choices where start(C)<=hLen and start(C)>prefLen do useful ← [C]++useful;
    // A tail choice is "useful" if, and only if, it contains, either in the ContAfter
       component or in the MinMax component, a requirement not yet satisfied by prefInc
    for C in choices where start(C)==hLen+1 do
        if exists ContAfter in ContAfterOf(C) where prefInc[ContAfter]=0 then
            useful ← [C]++useful;
        else if exists MinMax in MinMaxOf(C) where prefInc[MinMax] < min(MinMax) then
            useful ← [C]++useful;
    return useful;
// return a copy of prefInc, where the incidence of all assertions in C has been incremented
updateIncAfterChoice(prefInc, C)
    newPrefInc = copy (prefInc) ;
    for assertion in (ContAfterOf(C)) ++ MinMaxOf(C)) do newPrefInc[assertion] ← newPrefInc[assertion]+1;
    return newPrefInc;
// returns true if some upper bound in aList is violated by the incidence list prefInc
maxViolated(aList, prefInc)
    return (exists assert in aList where assert matches MinMax and prefInc[assert] > max(assert))
```

Definition 29 (useful choice). Choice $([i,.],.,AP',KP',.)$ is *useful* for assertions $AP'' \cup KP''$ iff

$$((AP' \cup KP') \cap (AP'' \cup KP'')) \neq \emptyset.$$

We can now describe our generation algorithm (Algorithm 3).

Given an array group $S = \{\,\mathsf{type(Arr)}, IP, AP, KP\,\}$ and a sound assignment A, we collect all choices C where $A(var(C))$ is not empty in *wChoices*: these are the choices having a witness in A. We now explore the space of all solutions for S that can be built using *wChoices*, with the recursive function *cList*.

cList is invoked with parameters *(hLen, aList, wChoices, prefLen, prefInc)*, where *hLen=HL*$(IP \cup AP)$ is the head length of S, *aList=*$AP \cup KP$ is the union of the AP' and KP' components of S, and *wChoices* are the choices that are witnessed in A. Each recursive call of *cList* assumes that a "prefix" of the solution has already been computed, where this prefix is a list of choices with length *prefLen*, whose incidence on the assertions in *aList* is reported in *prefInc*; the outermost call to *cList* sets *prefLen* to 0 and *prefInc* to a function that maps every assertion to 0. Assuming that the prefix exists,

the algorithm finds a well-formed choice list $\mathbb{C}$ such that the concatenation of a prefix characterized by *prefLen* and *prefInc* with $\mathbb{C}$ is a solution for $\{\,\mathsf{type(Arr)}, IP, AP, KP\,\}$.

If *aList=AP* $\cup$ *KP* is already satisfied by *prefInc*, then *cList* returns the empty choice list (line 2): adding an empty list to the prefix yields a solution. Otherwise, we first remove some choices that are not useful (line 3), so as to obtain *uWChoices*, the set of useful witnessed choices; this step is fundamental in order to avoid infinite recursion while exploring the tail of the solution, as proved in the generativity proof (Property 18). Then, we try to add a choice for position *prefLen+1* and, to this end, for each C in *uWChoices* whose interval includes position *prefLen+1*, we try to solve the subproblem *cList(hLen, aList, uWChoices, prefLen+1, newPrefInc)*, where *newPrefInc* is the incidence updated to keep track of the insertion of C. If one C exists such that *newPrefInc* does not violate any $I(\mathsf{cont}^M_m(x)) \leq M$ constraint and such that *cList(hLen, aList, uWChoices, prefLen+1, newPrefInc)* returns a solution $\mathbb{C}$ for the tail after C, then we return $[C]{+}{+}\mathbb{C}$ (line 9). If *uWChoices* contains no choice C such that *cList(hLen, aList, uWChoices, prefLen+1, newPrefInc)* has a solution, then we return "NoSolution".

This algorithm is sound and generative, and it always terminates.

Property 18 (Soundness and generativity). The algorithm *cList* is sound and generative, and terminates.

Proof. The algorithm terminates since the depth of recursion is limited by the fact that, at each step, we only keep the useful choices in the tail (Definition 29), hence, every choice that is selected after we arrive at *prefLen=hLen+1*, either (a) increments to one the incidence over an assertion $\mathsf{contAfter}(i^+ : x)$ whose incidence was zero, or (b) increments by one the incidence over an assertion $\mathsf{cont}^M_m(x)$ whose incidence was still below m, hence the depth of recursion is never greater than *MaxSteps*:

$$MaxSteps \;=\; h + |AP| + \Sigma_{\mathsf{cont}^M_m(x) \in KP}\; m.$$

Here, h is the head-length, $|AP|$ is an upper bound for the (a) steps, and $\Sigma_{\mathsf{cont}^M_m(x) \in KP}\; m$ is an upper bound for the steps of type (b). Hence, the algorithm is guaranteed to terminate.

For generativity, assume that an array group $S = \{\,\mathsf{type(Arr)}, IP, AP, KP\,\}$ with head-length h has a witness $J = [J_1, \ldots, J_o]$ with depth $d+1$, and assume that A is d-complete for E. For every i of $\{1..o\}$, we define $AP(i)$ and $KP(i)$ to be the sets of assertions in AP and in KP whose variable is inhabited by J_i, as follows:

$$\begin{array}{lcl} AP(i) & = & \{\,S \mid\; S \in AP,\; S = \mathsf{contAfter}(l^+ : x),\; i > l,\; J_i \in [\![x]\!]_E\,\} \\ KP(i) & = & \{\,S \mid\; S \in KP,\; S = \mathsf{cont}^M_m(x),\; J_i \in [\![x]\!]_E\,\}. \end{array}$$

As a consequence, $J_i \in [\![co(x)]\!]_E$ for any $\mathsf{cont}^M_m(x) \in (KP \setminus KP(i))$. We now build a witness J' by deleting some "useless" elements from J, and a choice list $\mathbb{C}'$ that describes J', using the following algorithm.

We initialize a variable $\mathbb{C}'$ with the empty list of choices. If $\mathbb{C}'$ is already satisfies AP and KP, then we set $J' = []$, and we terminate. Otherwise, we let l indicate the next position of $\mathbb{C}'$ to be defined, i.e., $l = |\mathbb{C}'| + 1$, and we consider the choice $([l, .], ., AP(l), KP(l), .)$. We say that the choice $([l, .], ., AP(l), KP(l), .)$ is useful for $AP \cup KP$ if either $l \leq h$ where h is the head length of S, or if $l > h$ and the choice contains some requirements from $AP \cup KP$ that are not yet satisfied by an array that is described by $\mathbb{C}'$ (this notion corresponds to the test performed by the function *usefulChoices* in Algorithm 3). If $l \geq h+1$ and $([l, .], ., AP(l), KP(l), .)$ is not a useful choice for $AP \cup KP$, then we can delete J_l from J', and what we obtain is still a witness: J_l does not contribute to the satisfaction of the requirements that are not yet satisfied, and the fact that all elements after J_l decrease their position by 1 is irrelevant since we are in the tail (Property 15). If we are not in the

tail, or we are in the tail and $([l, .], ., AP(l), KP(l), .)$ is a useful choice, then we leave J_l in J', we put $([\min(h+1, l), .], ., AP(l), KP(l), .)$ in $\mathbb{C}'$, and we continue.

At the end of this process, we have a new witness J', obtained by deleting some elements from J, and a corresponding choice list $\mathbb{C}' = [C'_1, \ldots, C'_p]$. Let us define $AP'(i)$ and $KP'(i)$, with $i \in \{1..p\}$, as the AP and KP components of C'_i, so that $C'_i = ([\min(h+1, i), .], ., AP'(i), KP(i), .)$, and it corresponds to J'_i in the same way as $([\min(h+1, i), .], ., AP(i), KP(i), .)$ corresponds to J_i.

We show now that every J'_i belongs to $[\![x]\!]_E$ for all variables x that appear positively in the expression $s([i, .], ., AP'(i), KP'(i), .)$ and belongs to $[\![co(x)]\!]_E$ for all complements $co(x)$ that appear in the same expression. In detail, the positive variables in the conjunction $s([i, .], ., AP'(i), KP'(i), .)$ come either from its derived component $IP' = IP \cap [i, i/\infty]$, or from $AP'(i)$, or from $KP'(i)$. For those variables x that come from assertions in $AP'(i)$ or $KP'(i)$, $J'_i \in [\![x]\!]_E$ is an immediate consequence of the way we defined $AP'(i)$ and $KP'(i)$. For the variables $co(x)$ that come from the implicit fifth element $KP \setminus KP'(i)$ of the choice, $J'_i \in [\![co(x)]\!]_E$ is a consequence of the fact that we have put in $KP(i)$ all the assertions where $J_i \in [\![x]\!]_E$, hence for all assertions in $KP \setminus KP(i)$ we have that $J_i \in [\![co(x)]\!]_E$. For the variables x that come from the derived component IP', we observe that J' is a witness for S, hence J'_i satisfies all constraints in IP that apply to its position, hence J' belongs to $[\![x]\!]_E$ for all variables x for which an applicable constraint $\mathsf{item}(j : x)$ or $\mathsf{items}(j^+ : x)$ belongs to IP'.

Since J'_i belongs to $[\![x]\!]_E$ for all variables x that appear positively in $s([i, .], ., AP'(i), KP'(i), .)$ and belongs to $[\![co(x)]\!]_E$ for all variables x that appear negated in $s([i, .], ., AP'(i), KP'(i), .)$, then it belongs to $[\![s([i, .], ., AP'(i), KP'(i), .)]\!]_E$, hence it also belongs to $[\![var([i, .], ., AP'(i), KP'(i), .)]\!]_E$. Since we assumed that J has depth $d+1$, then every J'_i has depth d or smaller, hence, for any A that is d-complete, every variable $var([i, .], ., AP'(i), KP'(i), .)$ in the list $\mathbb{C}'$ is *Witnessed* in A. Hence, the choice list $\mathbb{C}'$ that we have built is a list of choices that are *Witnessed* in A, and is such that every tail choice C in $\mathbb{C}'$ is "useful" according to the criterion used in Algorithm 3; we can conclude that the choice list $\mathbb{C}'$ would be generated by our algorithm unless a different solution were generated before, hence our algorithm is generative.

Soundness of the algorithm is immediate, since *cList* verifies that the generated solution satisfies all constraints and all requirement of the array group. □

Property 19 (Complexity). For any array group whose size is in $O(N)$, the algorithm *cList* has a complexity in $O(2^{poly(N)})$.

Proof. As proved in the proof of Theorem 18, the depth of the algorithm recursion is at most:

$$MaxSteps \;=\; h + |AP| + \Sigma_{\mathsf{cont}^M_m(x) \in KP}\; m.$$

By the small constants assumption, *MaxSteps* is in $O(N^2)$. At each level of the recursion, the `for` loop explores at most $O(2^{poly(N)})$ choices, hence the algorithm explores at most $O((2^{poly(N)})^{N^2}) = O(2^{poly(N)*N^2})$ choice lists, and the operations that must be executed for each choice list can be performed in time $O(2^{poly(N)})$, hence the total time is in $O(2^{poly(N)})$. □

5.5 Witness Generation from Base Typed Groups

Witness generation for groups with a base type does not need preparation and is fully accomplished during the first pass. We describe below the algorithm that we use for strings and numbers; the Null and Bool cases are trivial.

5.5.1 Type Str. A canonical group of type Str is just the conjunction of zero or more extended regular expressions, which we reduce to one by computing their intersection, whose size is linear

in the size of the input regular expressions. At this point, we generate a witness for this regular expression, which can be done in time $O(2^{poly(N)})$ (Section 3.5).

5.5.2 Type Num. For a canonical schema of type Num, we adopt the following "number generation algorithm". We first merge all intervals into one and all $\mathsf{mulOf}(m)$ operators into one, let us call it $\mathsf{mulOf}(M)$; if the group contains an assertion $\mathsf{notMulOf}(n)$ with $M = n \times i$ for any integer i, then the group returns "unsatisfiable". Otherwise, we obtain one interval (if none is present, we add $\mathsf{betw}_{-\infty}^{\infty}$), a set of zero or many $\mathsf{notMulOf}(n)$ constraints, and one optional $\mathsf{mulOf}(m)$ with $m \neq n \times i$ for every $i \in \mathbb{Z}$ and for every $\mathsf{notMulOf}(n)$. At this point, to simplify some operations, we substitute any negative argument n of $\mathsf{mulOf}(n)$ or $\mathsf{notMulOf}(n)$ with its opposite. The interval may be open at both extremes, closed at both, or mixed. We distinguish five cases. In the last three cases, we describe an open interval $\mathsf{xBetw}_{min}^{Max}$, but the reasoning when one extreme, or both, are included, is essentially the same.

(1) *Empty interval*: we return "unsatisfiable".
(2) *One-point interval* betw_{m}^{m}: if m satisfies all notMulOf and mulOf assertions we return m, otherwise we return "unsatisfiable".
(3) *Many-points interval* $\mathsf{xBetw}_{min}^{Max}$ *with no* $\mathsf{mulOf}(m)$ *constraint*: we have l $\mathsf{notMulOf}(n_j)$ constraints, with $l \geq 0$; choose ϵ such that

$$0 < \epsilon \leq \frac{min((Max - min), n_1, \ldots, n_l)}{l^+} \qquad \text{where} \qquad l^+ = 10^{\lceil \log_{10}(l+2) \rceil}.$$

If we consider the set $B = \{\, min + i \times \epsilon \mid i \in \{1..(l+1)\} \,\}$, then every value in B satisfies $\mathsf{xBetw}_{min}^{Max}$, and no assertion $\mathsf{notMulOf}(n_j)$ can be violated by two distinct values in B, hence at least one value in B is a witness.
(4) *Finite-width many-points interval* $\mathsf{xBetw}_{min}^{Max}$ *with a* $\mathsf{mulOf}(m)$ *constraint*: we have finite values for both *min* and *Max*: we list all multiples of m starting from *min* (excluded in case of xBetw) until we find one that satisfies all notMulOf assertions, or until we go over *Max* (excluded or included depending on the interval), in which case we return "unsatisfiable".
(5) *Infinite-width many-points interval* $\mathsf{xBetw}_{min}^{Max}$ *with a* $\mathsf{mulOf}(m)$ *constraint*: either *min* or *Max* is not finite, and we have a $\mathsf{mulOf}(m)$ constraint: JSON numbers are decimal number, hence we first rewrite all arguments of $\mathsf{mulOf}(m)$ and $\mathsf{notMulOf}(n)$ as integer fractions that all share the same decimal denominator d, as in $\mathsf{mulOf}(M/d)$, $\mathsf{notMulOf}(n_j/d)$. Select any prime number p that is strictly bigger than every n_j and such that either $p \times M/d$ or its opposite belongs to the interval; such a number exists since the interval is infinite in at least one direction, and it is easy to prove that primality of p and the fact that $(M/d) \neq (n_j/d) \times i$ for every $i \in \mathbb{Z}$ and for every $\mathsf{notMulOf}(n_j/d)$ imply that $p \times M/d$ satisfies all notMulOf assertions (Property 20, proof in the Supplementary Material).

Property 20. If a group of type Num has a witness, then the number generation algorithm returns a witness, otherwise it returns "unsatisfiable".

Property 21. The number generation algorithm terminates in time $O(2^{poly(N)})$.

6 Experimental Analysis

Our goal is the definition and implementation of an algorithm that is correct and complete for Classical JSON Schema without `uniqueItems` and that is fast enough to be used for real schemas, despite the exponential complexity of the problem.

Our algorithm can be used for witness generation as well as for containment checking. The asymptotic cost of containment checking is not greater than that of witness generation, since $\mathcal{S}_1 \subseteq \mathcal{S}_2$ holds iff $\mathcal{S}_1 \wedge \neg\mathcal{S}_2$ has no witness. Yet in practice, containment checking is much more expensive:

most real-world schemas present a very limited nesting of disjunction inside conjunctions and make little use of patterns in object types, which means that the heaviest phases of our witness-generation algorithm, which are DNF normalization and object/array preparation, can be expected to run in a reasonable time on such schemas. On the other hand, long conjunctions inside a disjunction are common. When evaluating $\mathcal{S}_1 \wedge \neg\mathcal{S}_2$, the not-elimination in $\neg\mathcal{S}_2$ may create long disjunctions inside conjunctions, hence a massive expansion of the schema during DNF normalization. Moreover, the conjunction of normalized $\neg\mathcal{S}_2$ with $\mathcal{S}_1$ triggers a new phase of DNF normalization and, when $\mathcal{S}_1$ and $\mathcal{S}_2$ are similar, the schemas $\mathcal{S}_1$ and $\neg\mathcal{S}_2$ present the same object fields with opposite types, which requires a lot of elaboration to prepare the resulting schema for the generation phase. Hence, the ability of our algorithm to generate witnesses in a reasonable time for real-world schemas would not imply the same result for containment checking.

In our previous work [10, 11], we focused our experiments on witness generation, with a strong emphasis on real-world schemas, and presented only one containment experiment on small synthetic schemas. In the following, we enrich that work and add four new containment experiments; two are still based on artificial schemas, but two are based on real-world schemas.

6.1 Research Hypotheses

We test the following hypotheses regarding witness generation: (H1) *correctness* of our implementation, that we test with the help of an external tool that validates the generated witnesses; (H2) *completeness* of our implementation, with respect to the language operators, that we test by using an ample and diverse test-set; (H3) *advancement of the state-of-the-art*, that we test by comparing our tool with state-of-the-art tools; (H4) *acceptable runtime* of our implementation on sizable real-world schemas, despite its asymptotic complexity of the algorithm, which we test by applying our tool to a vast set of real-world schemas.

In this version of the article, we add two hypotheses related to containment and equivalence: (H5) *utility for quality control of other tools*: a tool that performs schema rewriting can use our algorithm to verify that a rewritten schema is actually equivalent to the input schema. This extra check can be useful to ensure the quality of the rewriting tool. We test this hypothesis on the open source tool [25] used for merging *allOf* expressions. (H6) *acceptable runtime for containment checking on real-world schemas*, that we test on sizable real-world schemas.

6.2 Implementation and Experimental Setup

Our implementation uses in Java 11, with the Brics library [31] to generate witnesses from patterns, and the *jdd* library [37] for ROBDDs. Our environment is a 40-core Debian server with 384 GB of RAM. Each core runs with 3.1 Ghz.

We set the JVM heap size to 10 GB, reflecting a typical user configuration. All witnesses were validated by an external tool [1] and also by hand, as the external tool reported some false negatives. All reported times are measured for a single run. Per schema, we enforce a 10-minute timeout.

6.3 Tools for Comparative Experiments

Lacking equivalent tools, we compare against a data generator and a containment checker.

Data generator (DG). We use an open source test data generator for JSON Schema [16] (version 0.4.7), written in Java. It proceeds in try-and-fail approach: a generated example is validated against the schema. If validation fails, the example can be refined based on the error message. This tool lends itself to a comparison in witness generation experiments, yet it is not able to detect schema unsatisfiability: given an unsatisfiable schema, it will always return an (invalid) instance.

Containment checker (CC). We compare our tool against the containment checker described by Habib et al. [23] (version 0.0.5), designed to check interoperability of data transformation

Table 3. Overview of Schema Collections: Real-world Schemas (1st Group) and Test Schemas (2nd Group)

Collection	Origin	#Total	#Sat	#Unsat	Avg Size (KB)	Max Size (KB)
GitHub schemas	[13]	6,427	6,387	40	8.5	1,145.0
Kubernetes	[28]	1,092	1,087	5	23.8	1,310.7
Snowplow	[6]	420	420	0	3.8	54.8
WashingtonPost	[34]	125	125	0	20.9	141.7
Handwritten WG	[11]	235	197	38	1.5	109.4

Table 4. Overview of Schema Collections: Test Schemas (1st Group) and Real-world Schemas (2nd Group)

Collection	Origin	#Total	#$\not\subseteq$	#$\subseteq$	#Unknown	Avg Size (KB)	Max Size (KB)
Synthesized SC	[9]	1,331	450	881	0	0.5	2.9
MergeAllOf	[25]	174	7	167	0	0.6	2.1
Handwritten SC		282	118	164	0	0.8	3.9
SchemaStore vers.	[3]	1056	*144*	*871*	*41*	28.7	938.4
CC-testset	[24]	300	192	108	0	32.4	511.8

operators [15]. Typically, these schemas do not contain negation or recursion. The CC tool only supports Draft-04 schemas, a limitation that we consider in our discussion.

6.4 Schema Collections for Witness Generation

We perform witness generation experiments with the schema collections described in Table 3.

Real-world schemas. We used four collections of real-world schemas. The first and largest [13] has been created by collecting all files from GitHub that present features of JSON Schema, and performing rigorous duplicate elimination. In this collection, we identified, using our tool and by a further manual inspection, 40 schemas that are not satisfiable.

The remaining collections comprise standards for deploying applications (Kubernetes [28]), ruling interactions within a specific system (Snowplow [6]), and describing data produced by content management systems (Washington Post [34]). To increase the number of processable schemas, we inline references to external schemas. An earlier version of these three collections was also used by Habib et al. [23] to check inclusion. These schemas are satisfiable, except for 5 from Kubernetes.

Handwritten schemas for witness generation (Handwritten WG). Real-world schemas reflect real usage and can be quite large, yet are biased toward common operators and combinations of operators. For stress-testing, we include 235 handwritten schemas that are small, but that exemplify complex interactions: For objects, we test interactions among props, between props and req, and also between these operators and pro_i^j, for example in situations where there are at most n different field names that match a combination of patterns and there is a $\mathsf{pro}_{n+1}^{\infty}$ assertion; for arrays, we test the interactions among $\mathsf{item}(l : S)$ and $\mathsf{items}(i^+ : S)$, and also between these operators and $\mathsf{cont}_i^j(S)$; for strings, we basically test the interaction between patterns (pattern) and the lower/upper-bound for the length of string, which, in the algebra, is captured by ad hoc patterns; for numbers, we test the interaction among combinations of betw_m^M and xBetw_m^M, $\mathsf{mulOf}(q)$, and $\mathsf{notMulOf}(q)$; we test the behavior of the same combinations under negation and in presence of recursion.

6.5 Schema Collections for Containment Checking

We perform containment check experiments with the schema collections described in Table 4.

Synthesized Schema Containment. We include a published test suite for checking JSON Schema containment [9]. This collection of small schema pairs, with a ground truth regarding schema containment, was synthesized from the official JSON Schema Test Suite [32]. The original test suite was designed by the JSON Schema group in order to cover all language operators, and the synthesized tests inherits this property; we excluded schemas that contain features that we do not yet support, such as the `format` keyword (a mere technicality) or references to external files.

MergeAllOf containment. The open source JavaScript tool *json-schema-merge-allof* [25] simplifies schemas by merging `allOf`branches. To test hypothesis H5, we extracted the unit tests published with the tool. These rewrite a schema $\mathcal{S}_1$ and compare it to the expected result schema $\mathcal{S}_2$. From each unit test, we produced the test cases $\mathcal{S}_1 \subseteq \mathcal{S}_2$ and $\mathcal{S}_2 \subseteq \mathcal{S}_1$, which should both hold given that the tool should transform a schema into an equivalent one.

Handwritten for Schema Containment (Handwritten SC).. As in witness generation, we designed handwritten tests. For breadth, we systematically considered different pairings of structural or Boolean operators with other structural or Boolean operators, in the alternation between containment and non-containment cases, and in considering the different reasons why containment may hold. For depth, we focus on scenarios where containment holds because of complex interactions between different operators; the Supplementary Material (Appendix D) contains an example.

SchemaStore Versions. A use-case for a containment tool is that a new version $\mathcal{S}'$ of a schema $\mathcal{S}$ is released, and one needs to verify whether $\mathcal{S}'$ is backward compatible, in the sense that every instance accepted by $\mathcal{S}$ is also accepted by $\mathcal{S}'$ ($\mathcal{S} \subseteq \mathcal{S}'$). We test hypothesis H6 by comparing pairs of successive versions of schemas published on SchemaStore. We can access historic schema versions from the backing GitHub repository. We exclude all schemas that contain references or pattern constructs that we do not support (lookahead and lookbehind). In addition, we rename the occurrences of `uniqueItems` and `format` so that they are ignored.

For these schema pairs, we have no *a-priori* knowledge of the correct result; hence, in Table 4, we classify the schemas according to the result returned by the experiments, which explains the non-zero entry for "unknown". Since we lack a ground truth, this experiment is only used to test hypothesis H6 regarding runtime, but not hypothesis H1 regarding correctness.

CC-testset. The authors of the CC tool [23] provide a replication package for their experiments, which includes 300 pairs of schemas with a ground truth w.r.t. containment. This collection contains schemas from Kubernetes, Snowplow, and WashingtonPost, but also schemas from the use case that motivated their tool. Again, we rename `format` keywords.

6.6 Experimental Results

We run witness generation experiments, where we compare our tool with the DG tool, and experiments about containment-checking, where we compare our tool with the CC tool. The comparison is not intended as a "competition", since the tools have been designed to address different needs. Specifically, our tool is the only one that has been designed with completeness as a primary concern. We compare our tool with the state-of-the-art to test hypothesis H3, that is, to dispel the possibility that existing tools, despite the fact that they have not been designed for completeness, are already good enough to render our tool redundant.

We distinguish four outcomes: *success*, when a result is returned and it is correct; *interruption*: when the code raises an "unsupported" exception, a run-time error, or a timeout; *logical error on satisfiable schema*, when the input schema $\mathcal{S}$ is satisfiable but the code returns either "unsatisfiable" or a witness that does not actually satisfy $\mathcal{S}$; *logical error on unsatisfiable schema*, when the input schema is unsatisfiable but a presumed witness is nevertheless returned.

Table 5. Schema Collections, Correctness and Completeness, Median/95th Percentile/Average Runtime (in sec.)

Collection	Tool	Success	Interrupt.	Errors sat	Errors unsat	Med. Time	95% -tile	Avg. Time
GitHub	Ours	98.96%	1.04%	0%	0%	0.021	0.637	1.104
witness	DG	92.9%	4.19%	2.47%	0.44%	0.018	0.065	0.174
Kubernetes	Ours	100%	0%	0%	0%	0.013	0.49	0.56
witness	DG	99.54%	0%	0%	0.46%	0.018	0.08	0.028
Snowplow	Ours	99.52%	0.48%	0%	no unsat	0.04	1.974	1.122
witness	DG	94.76%	0%	5.24%	no unsat	0.019	0.059	0.025
Washington Post	Ours	100%	0%	0%	no unsat	0.031	131.053	23.048
witness	DG	96.8%	0%	3.2%	no unsat	0.022	0.102	0.037
Handwritten WG	Ours	100%	0%	0%	0%	0.035	1.414	1.717
witness	DG	7.23%	34.47%	50.21%	8.09%	0.02	0.076	0.037
Synthesized	Ours	100%	0%	0%	0%	0.004	0.037	0.01
containment	CC	35.91%	62.96%	0.15%	0.98%	0.003	0.11	0.018
MergeAllOf	Ours	100%	0%	0%	0%	0.017	0.089	0.035
containment	CC	45.4%	42.53%	0.57%	11.49%	0.01	0.059	0.171
Handwritten SC	Ours	100%	0%	0%	0%	0.024	0.092	0.037
containment	CC	40.43 %	38.3 %	7.8 %	13.48 %	0.028	1.032	0.43
SchemaStore vers.	Ours	86.93%	13.07%	0%	0%	0.356	15.042	9.271
containment	CC	79.07%	20.93%	0%	0.09%	0.033	2.773	4.353
CC-testset	Ours	81.33%	18.67%	0%	0%	0.127	10.751	2.16
containment	CC	92.67%	7.33%	0%	0%	0.086	7.032	0.981

We summarize the results of the experiments in Table 5 and discuss them below.

6.6.1 Correctness and Completeness. Our tool does not produce logical errors in any of our schema collections. For the GitHub schemas, witness generation fails for 64 schemas. In about a third of the cases, this is caused by an "out of memory" error when calling the automata library, and a timeout otherwise. We further observe 2 timeouts for the Snowplow collection. In containment checking, we observe 138 timeouts for the SchemaStore collection. In particular, we observe only one timeout for the CC tool with this collection. No interruptions arise in the other schema collections.

We have no logical error, and no interruption is related to an incomplete coverage of the JSON Schema language apart from external references, hence supporting hypothesis H1 (correctness) and H2 (completeness), in the sense of covering all features of the language.

The DG and CC tools have a lower success rate for all schema collections but one. They display a higher distance on datasets designed to test completeness, such as the two handwritten datasets, and the one synthesized from the JSON Schema test suite, but also with respect to the test cases of the MergeAllOf tool. For the CC tool, the high percentage of "interruptions" should not be over-emphasized, since it is just due to unsupported features. However, the CC tool also shows a non-negligible amount of logical errors on many datasets. We can conclude that our tool advances the state-of-the-art for witness generation and containment checking, especially from the viewpoint of correctness and language coverage for schemas that present aspects of complexity (hypothesis H3).

Note that for the CC-testset collection, our success rate is less than that of the CC tool. In half of the cases, interruptions of our tool are due to schemas using external references, a technical limitation. Yet the other half of the cases are schemas with `minItems` values in the range of tens of thousands. This leads to very large witnesses.

6.6.2 Generation Runtime on Real-world Schemas. We next analyze the crucial hypothesis H4. In the three largest collections, 95% of the files are processed in less than 2 secs, with median ≤5 ms, and average ≤1.5 secs. These results are coherent with hypothesis H4. The smaller Washington Post collection presents higher runtimes and will be discussed in Section 6.7.

6.6.3 Containment-checking Runtime on Real-world Schemas. The experiment on SchemaStore versions and CC-testset was primarily intended to test hypothesis H6. The above 80% success rate indicates that the current implementation compares well with the state-of-the-art in terms of coverage, but it is very far from the near-99% rate of witness generation experiments. For the CC-testset collection, half of the interruptions are due to schemas containing external references. The 0.4 seconds of median time and the ca. 15 seconds of the 95%-tile may be acceptable for a final check before deploying a new version of a schema, but the 95%-tile value is too high for other use cases.

For SchemaStore versions, the CC tool fails on 221 cases (with one timeout); our tool successfully analyzes 171 cases, confirming the hypothesis that it advances the state-of-the-art. However, the median time and the 95%-tile times are one order of magnitude smaller for the CC tool.

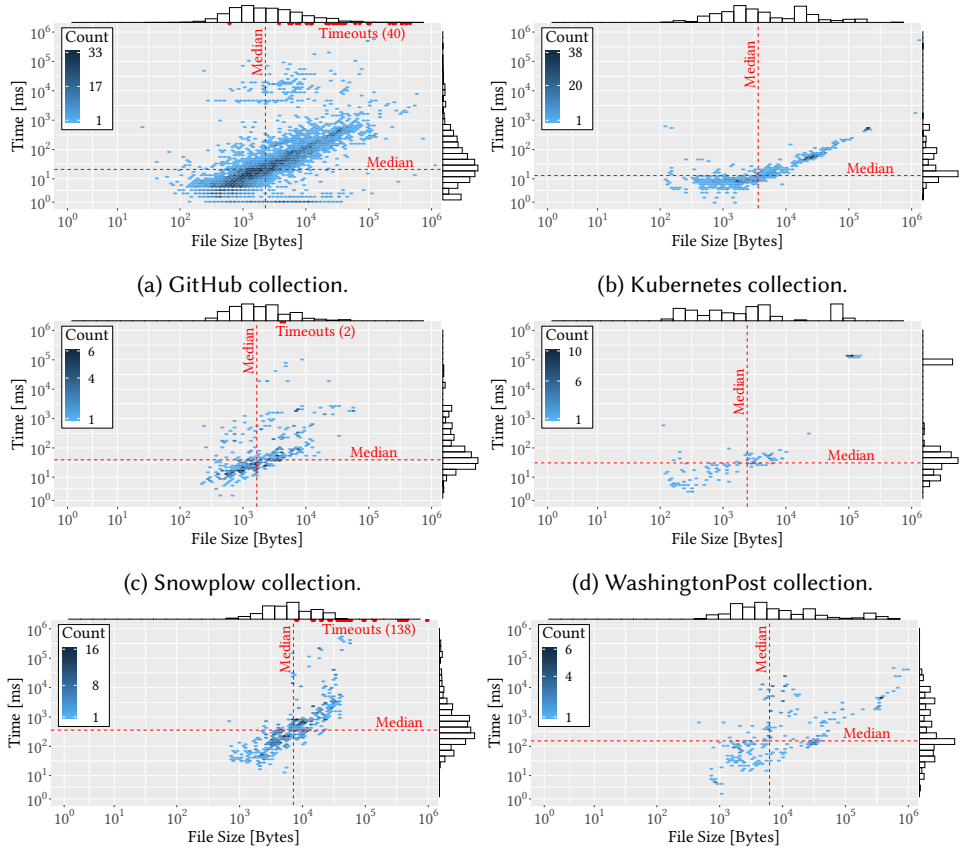
For the CC-testset collection, our tool shows a relative runtime comparable to the SchemaStore versions collection, the CC tool being slightly faster. Overall, the CC tool implements the standard approach of applying a set of incomplete deduction rules that take a short time and cover the majority of cases, while our tool uses a more complex but complete algorithm. It would be interesting to implement a tool that combines a fast rule-based first attempt with a fallback to the complete and slow algorithm when needed, but this is beyond the scope of this article.

Hence, hypothesis H6 is still open: the speed of inclusion-checking for real-world schemas of the current implementation is not fully satisfactory, but our implementation was not designed nor optimized for this specific use, and there is ample opportunity for focused optimization.

6.6.4 Quality Control for the MergeAllOf Tool. We tested hypothesis H5 by examining the ability of our tool to correctly verify all test cases published for the MergeAllOf tool. Our tool was able to correctly verify all 174 test cases with an average time in the sub-second range, with runtimes comparable to the CC tool, which has, however, only been able to verify 45% of the cases. This is fully satisfactory, but the added unexpected outcome was that our tool detected 7 cases where the MergeAllOf tool produced a schema that was *not* equivalent to the schema submitted to and-merge. Hence, our tool has actually been useful in discovering problems of the MergeAllOf tool. We have communicated our findings to the author of the MergeAllOf tool.

6.7 Qualitative Insights

The scatterplots in Figure 7 visualize the relationship between schema size and tool runtime. Several interesting insights can be extracted for the GitHub collection, shown in Figure 7(a). The histograms at the top and at the right-hand side indicate that schema size and run-time are distributed along several orders of magnitude, with a strong concentration on the low part of both axes, hence the log-log scale. We observe a cloud with a slope of about 1, suggesting a linear correlation, but we also observe that all file sizes exhibit outliers and that long-running schemas can be found everywhere along the file size axis. This clearly indicates that rather than the size of the schema, the runtime is affected more by the presence of specific combinations of operators.



(a) GitHub collection. (b) Kubernetes collection.

(c) Snowplow collection. (d) WashingtonPost collection.

(e) SchemaStore vers. collection. (f) CC-testset collection.

Fig. 7. Scatterplots showing size of the schema vs. runtime of our implementation. Stylized histograms show the distributions. Top right, the sizes of the files causing timeouts are shown, along with number of timeouts.

Our complexity analysis shows that exponential complexity is triggered by 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 align 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 $n > 65000$, which induces the creation of a large automaton. Other critical features are the presence of `"enum"` and `"oneOf"` with very long lists of arguments, which may cause the generation of very big terms during DNF reduction, and of nested objects with overlapping patterns, which may also require a lot of time, as indicated by the theoretical analysis.

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 ca. 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 31 ms. All the "slow" files are very similar, with more than 2K nodes in their syntax trees and complex combinations of operators. 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.

Runtime for the other collections is comparable to that of the GitHub collection. For the containment experiments, we observed a high percentage of timeouts; this was expected since containment checking is inherently more difficult than witness generation.

7 Conclusions

JSON Schema is widely used in data-centric applications. The decidability and complexity of satisfiability and containment were known, but no explicit algorithm had been defined, and it was not obvious whether the high asymptotic complexity of the problem was compatible with a practical algorithm. In this article, we have addressed this open problem. We have described an algorithm for witness generation, satisfiability, and containment, that is based on a specific combination of known and original techniques, to take into account the specific features of JSON Schema object and array operators, and the need to run in a reasonable time. 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.

The techniques we developed can also be extended to deal with the uniqueItems operator. To this aim, the algorithms we presented here must be extended to generate many witnesses for any schema, and a more complex termination condition for bottom-up generation must be adopted. The theme is important, but we cannot develop it here any further, for space reasons.

Nomenclature

Metavariable	Meaning	Page
e	An externally extended regular expression.	7
S	Any algebraic expression.	8
E	An environment.	8
r	A plain regular expression.	9
$\mathcal{S}$	A JSON Schema schema.	10
T, T'	Types.	17
$\mathcal{A}_E^i$	A sequence of assignments.	19
A^i	A finite assignment.	20
p, p_i	Patterns.	22
CP	Constraining part, i.e., the set of props assertions $\{\mathsf{props}(p_i : x_i) \mid i \in 1..m\}$.	22
RP	Requiring part, i.e., the set of pattReq assertions $\{\mathsf{pattReq}(e_j : y_j) \mid j \in 1..n\}$.	22
$\mathbb{C}$	A list of choices.	25
C, C_i, CL_i	Choices.	25
J	A JSON value.	26
a_i, k	Property names.	27
AP	A set of *contains-After* requirements with shape $\mathsf{contAfter}(i^+ : x)$.	28
KP	A set of *counting* assertions $\mathsf{cont}_i^j(x)$.	28
$\mathcal{S}$	A set of assertions.	30

Function Name	Meaning	Page
$co(x)$	Complement variable of variable x.	15
$dnf(\_)$	Normalization function.	16

(Continued)

$ITO(T)$	ITOs associated to type T.	17
$\langle\!\langle \_ \rangle\!\rangle_A$	Assignment evaluation function.	18
$Vars(E)$	Variables of an environment E.	18
$T_E(\_)$	Assignment transformation function.	18
$\delta(J)$	Depth of a JSON value J.	19
$gen(\_, \_)$	Function mapping a schema and an assignment into a set of JSON values.	20
$Vars(A)$	Variable defined in an assignment A.	20
$cp(CP', RP')$	The characteristic pattern of CP', RP'.	22
$I_{\mathbb{C}}(S)$	Incidence of $\mathbb{C}$ over S.	32
$AP(i)$	The set of assertions in AP whose variable is inhabited by a value J_i.	34
$KP(i)$	The set of assertions in KP whose variable is inhabited by a value J_i.	34

Acronym	**Meaning**	**Page**
ITO	Implicative Typed Operator.	7
TO	Typed Operator.	7
DNF	Disjunctive Normal Form.	13
ROBDD	Reduced Ordered Boolean Decision Diagram.	15

References

[1] 2022. JSON schema validator. Retrieved 19 September 2022 from https://github.com/networknt/json-schema-validator

[2] 2024. hypothesis-jsonschema. Available on GitHub. Retrieved 4 December 2024 from https://github.com/python-jsonschema/hypothesis-jsonschema

[3] 2024. JSON Schema Store. Retrieved 23 March 2026 from https://www.schemastore.org

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[5] Margareta Ackerman and Erkki Mäkinen. 2009. Three new algorithms for regular language enumeration. In *Proceedings of the International Computing and Combinatorics Conference*. Hung Q. Ngo (Ed.), Lecture Notes in Computer Science, Vol. 5609, Springer, 178–191.

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

[7] Henry Andrews. 2023. Modern JSON Schema. Retrieved 4 December 2024 from https://modern-json-schema.com/

[8] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Database Theory, Querying Data*. Preliminary Version.

[9] 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 *Proceedings of the ER 2021*. 19–24.

[10] 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 *Proceedings of the EDBT 2021*. 694–697. DOI:10.5441/002/edbt.2021.86 Tool Demo.

[11] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness generation for JSON schema. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4002–4014. Retrieved from https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf

[12] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of modern JSON schema: Formalization and complexity. In *Proceedings of the ACM on Programming Languages 8, POPL* (2024), 1451–1481.

[13] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023. Negation-closure for JSON schema. *Theoretical Computer Science* 955 (April 2023), 113823. DOI: https://doi.org/10.1016/j.tcs.2023.113823

[14] Fernando Suárez Barría. 2016. *Formal Specification, Expressiveness, and Complexity Analysis for JSON Schema*. Master's thesis. Pontificia Universidad Católica de Chile, Santiago, Chile. Retrieved from https://repositorio.uc.cl/handle/11534/16908

[15] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. Lale: Consistent automated machine learning. arXiv:2007.01977. Retrieved from https://arxiv.org/abs/2007.01977

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

[17] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 123–135. DOI: https://doi.org/10.1145/3034786.3056120
[18] Randal E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35, 8 (1986), 677–691. DOI: 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* (2008), 262 pages. Available online at https://hal.inria.fr/hal-03367725/file/tata.pdf.
[20] Dominik D. Freydenberger. 2013. Extended regular expressions: Succinctness and decidability. *Theory of Computing Systems* 53, 2 (2013), 159–193. DOI: https://doi.org/10.1007/s00224-012-9389-0
[21] Wouter Gelade, Wim Martens, and Frank Neven. 2007. Optimizing schema languages for XML: Numerical constraints and interleaving. In *Proceedings of the International Conference on Database Theory.* , Thomas Schwentick and Dan Suciu (Eds.), Lecture Notes in Computer Science, Vol. 4353, Springer, 269–283. DOI: https://doi.org/10.1007/11965893_19
[22] Wouter Gelade and Frank Neven. 2012. Succinctness of the complement and intersection of regular expressions. *ACM Transactions on Computational Logic* 13, 1 (2012), 4:1–4:19. DOI: https://doi.org/10.1145/2071368.2071372
[23] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding data compatibility bugs with JSON subschema checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 620–632. DOI: https://doi.org/10.1145/3460319.3464796
[24] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. jsonsubschema (Replication Package). Retrieved 4 December 2024 from https://zenodo.org/records/4729863
[25] Martin Hansen. 2023. json-schema-merge-allof. Retrieved 4 December 2024 from https://github.com/mokkabonna/json-schema-merge-allof, commit hash 133f848.
[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation, 3rd Edition*. Addison-Wesley.
[27] Shiva Jahangiri. 2021. Wisconsin benchmark data generator: To JSON and beyond. In *Proceedings of the International Conference on Management of Data*. ACM, 2887–2889. DOI: https://doi.org/10.1145/3448016.3450577
[28] Kubernetes. 2022. Kubernetes JSON Schemas. Retrieved 4 December 2024 from https://github.com/instrumenta/kubernetes-json-schema, commit hash b3cf311.
[29] Wim Martens, Frank Neven, and Thomas Schwentick. 2009. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing* 39, 4 (2009), 1486–1530. DOI: https://doi.org/10.1137/080743457
[30] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. 2006. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems* 31, 3 (2006), 770–813. DOI: https://doi.org/10.1145/1166074.1166076
[31] Anders Møller. 2021. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. Available at Retrieved 19 September 2022 from https://www.brics.dk/automaton/
[32] JSON Schema Org. 2022. JSON Schema Test Suite. Retrieved 19 September 2022 from https://github.com/json-schema-org/JSON-Schema-Test-Suite
[33] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*. 263–273.
[34] The Washington Post. 2022. ans-schema. Retrieved 19 September 2022 from https://github.com/washingtonpost/ans-schema, commit hash abdd6c211.
[35] Helmut Seidl. 1990. Deciding equivalence of finite tree automata. *SIAM Journal on Computing* 19, 3 (1990), 424–437.
[36] Larry J. Stockmeyer. 1974. *The Complexity of Decision Problems in Automata Theory and Logic*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[37] Arash Vahidi. 2020. JDD. Retrieved 19 September 2022 from https://bitbucket.org/vahidi/jdd/src/master/
[38] 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.
[39] A. Wright, H. Andrews, and B. Hutton. 2020. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-bhutton-json-schema-validation-00*. Technical Report. Internet Engineering Task Force.
[40] 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.

Received 1 July 2024; revised 14 November 2025; accepted 6 February 2026