# Validating Streaming JSON Documents with Learned VPAs*

Véronique Bruyère[1], Guillermo A. Pérez[2], and Gaëtan Staquet[1,2](✉)

[1] University of Mons (UMONS), Mons, Belgium
{veronique.bruyere,gaetan.staquet}@umons.ac.be
[2] University of Antwerp (UAntwerp) – Flanders Make, Antwerp, Belgium
guillermo.perez@uantwerpen.be

**Abstract.** We present a new streaming algorithm to validate JSON documents against a set of constraints given as a JSON schema. Among the possible values a JSON document can hold, objects are unordered collections of key-value pairs while arrays are ordered collections of values. We prove that there always exists a visibly pushdown automaton (VPA) that accepts the same set of JSON documents as a JSON schema. Leveraging this result, our approach relies on learning a VPA for the provided schema. As the learned VPA assumes a fixed order on the key-value pairs of the objects, we abstract its transitions in a special kind of graph, and propose an efficient streaming algorithm using the VPA and its graph to decide whether a JSON document is valid for the schema. We evaluate the implementation of our algorithm on a number of random JSON documents, and compare it to the classical validation algorithm.

**Keywords:** Visibly pushdown automata · JSON · streaming validation

## 1 Introduction

*JavaScript Object Notation* (JSON) has overtaken XML as the de facto standard data-exchange format, in particular for web applications. JSON documents are easier to read for programmers and end users since they only have arrays and objects as structured types. Moreover, in contrast to XML, they do not include named open and end tags for all values, but open and end tags (braces actually) for arrays and objects only. *JSON schema* [13] is a simple schema language that allows users to impose constraints on the structure of JSON documents.

In this work, we are interested in the *validation* of *streaming* JSON documents against JSON schemas. Several previous results have been obtained about the formalization of XML schemas and the use of formal methods to validate XML documents (see, e.g., [5,15,16,18,24,25]). Recently, a standard to formalize JSON schemas has been proposed and (hand-coded) validation tools for such schemas can be found online [13]. Pezoa et al, in [19], observe that the standard

*This work was supported by the Belgian FWO "SAILor" project (G030020N). Gaëtan Staquet is a research fellow (Aspirant) of the Belgian F.R.S.-FNRS.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 271–289, 2023.
https://doi.org/10.1007/978-3-031-30823-9_14

of JSON documents is still evolving and that the formal semantics of JSON schemas is also still changing. Furthermore, validation tools seem to make different assumptions about both documents and schemas. The authors of [19] carry out an initial formalization of JSON schemas into formal grammars from which they are able to construct a *batch* validation tool from a given JSON schema.

In this paper, we rely on the formalization work of [19] and propose a *streaming* algorithm for validating JSON documents against JSON schemas. To our knowledge, this is the first JSON validation algorithm that is streaming. For XML, works that study streaming document validation base such algorithms on the construction of some automaton (see, e.g., [25], for XML). In [7], we first experimented with one-counter automata for this purpose. We submit that *visibly-pushdown automata* (VPAs) are a better fit for this task — this is in line with [15], where the same was proposed for streaming XML documents. In contrast to one-counter automata,[3] we show that VPAs are expressive enough to capture the language of JSON documents satisfying any JSON schema.

More importantly, we explain that *active learning à la* Angluin [4] is a good alternative to the automatic construction of such a VPA from the formal semantics of a given JSON schema. This is possible in the presence of labeled examples or a computer program that can answer membership and (approximate) equivalence queries about a set of JSON documents. This learning approach has two advantages. First, we derive from the learned VPA a streaming validator for JSON documents. Second, by automatically learning an automaton representation, we circumvent the need to write a schema and subsequently validate that it represents the desired set of JSON documents. Indeed, it is well known that one of the highest bars that users have to clear to make use of formal methods is the effort required to write a formal specification, in this case, a JSON schema.

*Contributions.* We present a VPA active learning framework to achieve what was mentioned above — though we fix an order on the keys appearing in objects. The latter assumption helps our algorithm learn faster. Secondly, we show how to bootstrap the learning algorithm by leveraging existing validation and document-generation tools to implement approximate equivalence checks. Thirdly, we describe how to validate streaming documents using our fixed-order learned automata — that is, our algorithm accepts other permutations of keys, not just the one encoded into the VPA. Finally, we present an empirical evaluation of our learning and validation algorithms, implemented on top of LearnLib [17].

All contributions, while complementary, are valuable in their own right. First, our learning algorithm for VPAs is a novel gray-box extension of TTT [9] that leverages side information about the language of all JSON documents. Second, our validation algorithm that uses a fixed-order VPA is novel and can be applied regardless of whether the automaton is learned or constructed from a schema. For the validation algorithm, we developed the concept of *key graph*, which allows us to efficiently realize the validation no matter the key-value order in the docu-

[3] By nesting objects and arrays, we obtain a set of JSON documents encoding $\{a^n b^m c^m d^n \mid n, m \in \mathbb{N}\}$, a context-free language that requires two counters.

ment, and might be of independent interest for other JSON-analysis applications using VPAs. Finally, we implemented our own batch validator to facilitate approximating equivalence queries as required by our learning algorithm. Both the new validator and the equivalence oracle are efficient, open-source, and easy to modify. We strongly believe the latter can be re-used in similar projects aiming to learn automata representations of sets of JSON documents.

A long version of this work is on arXiv: https://arxiv.org/abs/2211.08891.

## 2 Visibly Pushdown Languages

First, we recall the definition VPAs [3] and state some of their properties. We also recall how they can be actively learned following Angluin's approach [4].

**Visibly Pushdown Automata** An *alphabet* $\Sigma$ is a finite set whose elements are called *symbols*. A *word* $w$ over $\Sigma$ is a finite sequence of symbols from $\Sigma$, with the *empty word* denoted by $\varepsilon$. The length of $w$ is denoted $|w|$; the set of all words, $\Sigma^*$. Given two words $v, w \in \Sigma^*$, $v$ is a *prefix* (resp. *suffix*) of $w$ if there exists $u \in \Sigma^*$ such that $w = vu$ (resp. $w = uv$), and $v$ is a *factor* of $w$ if there exist $u, u' \in \Sigma^*$ such that $w = uvu'$. Given $L \subseteq \Sigma^*$, called a *language*, we denote by $\text{Pref}(L)$ (resp. $\text{Suff}(L)$) the set of prefixes (resp. suffixes) of words of $L$. Given a set $Q$, we write $\mathbb{I}_Q$ for the *identity relation* $\{(q, q) \mid q \in Q\}$ on $Q$.

VPA [3] are particular pushdown automata that we recall in this section. The *pushdown alphabet*, denoted $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_i)$, is partitioned into pairwise disjoint alphabets $\Sigma_c, \Sigma_r, \Sigma_i$ such that $\Sigma_c$ (resp. $\Sigma_r$, $\Sigma_i$) is the set of *call* symbols (resp. *return* symbols, *internal* symbols). In this paper, we work with the particular alphabet of return symbols $\Sigma_r = \{\bar{a} \mid a \in \Sigma_c\}$. For any such $\tilde{\Sigma}$, we denote by $\Sigma$ the alphabet $\Sigma_c \cup \Sigma_r \cup \Sigma_i$. Given a pushdown alphabet $\tilde{\Sigma}$, the set $\text{WM}(\tilde{\Sigma})$ of *well-matched* words over $\tilde{\Sigma}$ is defined:

- $\varepsilon \in \text{WM}(\tilde{\Sigma})$, and $a \in \text{WM}(\tilde{\Sigma})$ for all $a \in \Sigma_i$,
- if $w, w' \in \text{WM}(\tilde{\Sigma})$, then $ww' \in \text{WM}(\tilde{\Sigma})$,
- if $a \in \Sigma_c, w \in \text{WM}(\tilde{\Sigma})$, then $aw\bar{a} \in \text{WM}(\tilde{\Sigma})$.

Also, the *call/return balance* function $\beta : \Sigma^* \to \mathbb{Z}$ is defined as $\beta(\varepsilon) = 0$ and $\beta(ua) = \beta(u) + x$ with $x$ being 1, $-1$, or 0 if $a$ is in $\Sigma_c$, $\Sigma_r$, or $\Sigma_i$ respectively. In particular, for all $w \in \text{WM}(\tilde{\Sigma})$, we have $\beta(u) \geq 0$ for each prefix $u$ of $w$ and $\beta(u) \leq 0$ for each suffix $u$ of $w$. Finally, the *depth* $d(w)$ of a well-matched word $w$ is equal to $\max\{\beta(u) \mid u \in \text{Pref}(\{w\})\}$, that is, the maximum number of unmatched call symbols among the prefixes of $w$.

**Definition 1.** *A* visibly pushdown automaton *(VPA) over a pushdown alphabet $\tilde{\Sigma}$ is a tuple $(Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$ where $Q$ is a finite non-empty set of* states, *$Q_I \subseteq Q$ is a set of* initial *states, $Q_F \subseteq Q$ is a set of* final *states, $\Gamma$ is a* stack alphabet, *and $\delta$ is a finite set of* transitions *of the form $\delta = \delta_c \cup \delta_r \cup \delta_i$ where $\delta_c \subseteq Q \times \Sigma_c \times Q \times \Gamma$ is the set of* call *transitions, $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$ is the set of* return *transitions, and $\delta_i \subseteq Q \times \Sigma_i \times Q$ is the set of* internal *transitions. The* size *of $\mathcal{A}$ is denoted by $|Q|$, and its number of transitions by $|\delta|$.*

Let us describe the *transition system* $\mathrm{T}_{\mathcal{A}}$ of a VPA $\mathcal{A}$ whose vertices are configurations. A *configuration* is a pair $\langle q, \sigma \rangle$ where $q \in Q$ is a state and $\sigma \in \Gamma^*$ a stack content. A configuration is *initial* (resp. *final*) if $q \in Q_I$ (resp. $q \in Q_F$) and $\sigma = \varepsilon$. For $a \in \Sigma$, we write $\langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$ in $\mathrm{T}_{\mathcal{A}}$ if there is either a call transition $(q, a, q', \gamma) \in \delta_c$ verifying $\sigma' = \gamma\sigma$,[4] or a return transition $(q, a, \gamma, q') \in \delta_r$ verifying $\sigma = \gamma\sigma'$, or an internal transition $(q, a, q') \in \delta_i$ such that $\sigma' = \sigma$.

The transition relation of $\mathrm{T}_{\mathcal{A}}$ is extended to words in the usual way. We say that $\mathcal{A}$ *accepts* a word $w \in \Sigma^*$ if there exists a path in $\mathrm{T}_{\mathcal{A}}$ from an initial configuration to a final configuration that is labeled by $w$. The *language of* $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q \in Q_I, \exists q' \in Q_F, \langle q, \varepsilon \rangle \xrightarrow{w} \langle q', \varepsilon \rangle\}$, i.e., the set of all words accepted by $\mathcal{A}$. Any language accepted by some VPA is a *visibly pushdown language* (VPL). Notice that such a language is composed of well-matched words only.[5] Given a VPA $\mathcal{A}$ over $\tilde{\Sigma}$, the *reachability relation* $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ is $\mathrm{Reach}_{\mathcal{A}} = \{(q, q') \in Q^2 \mid \exists w \in \mathrm{WM}(\tilde{\Sigma}), \langle q, \varepsilon \rangle \xrightarrow{w} \langle q', \varepsilon \rangle\}$.

Finally, we say that $p \in Q$ is a *bin state* if there exists no path in $\mathrm{T}_{\mathcal{A}}$ of the form $\langle q, \varepsilon \rangle \xrightarrow{w} \langle p, \sigma \rangle \xrightarrow{w'} \langle q', \varepsilon \rangle$ with $q \in Q_I$ and $q' \in Q_F$. If a VPA $\mathcal{A}$ has bin states, those states can be removed from $Q$ as well as the transitions containing bin states without modifying the accepted language.

**Minimal Deterministic VPAs** Given a VPA $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$, we say that it is *deterministic* (det-VPA) if $|Q_I| = 1$ and $\mathcal{A}$ does not have two distinct transitions with the same left-hand side. By *left-hand side*, we mean $(q, a)$ for a call transition $(q, a, q', \gamma) \in \delta_c$ or an internal transition $(q, a, q') \in \delta_i$, and $(q, a, \gamma)$ for a return transition $(q, a, \gamma, q') \in \delta_r$.

**Theorem 1 ( [3,32]).** *For any VPA $\mathcal{A}$ over $\tilde{\Sigma}$, one can construct a det-VPA $\mathcal{B}$ over $\tilde{\Sigma}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Moreover, the size of $\mathcal{B}$ is in $\mathcal{O}(2^{|Q|^2})$ and the size of its stack alphabet is in $\mathcal{O}(|\Sigma_c| \cdot 2^{|Q|^2})$.*

*Proof.* Let us briefly recall this construction. Let $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$. The states of $\mathcal{B}$ are subsets $R$ of the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ and the stack symbols of $\mathcal{B}$ are of the form $(R, a)$ with $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and $a \in \Sigma_c$. Let $w = u_1 a_1 u_2 a_2 \ldots u_n a_n u_{n+1}$ be such that $n \geq 0$ and $u_i \in \mathrm{WM}(\tilde{\Sigma}), a_i \in \Sigma_c$ for all $i$. That is, we decompose $w$ in terms of its unmatched call symbols. Let $R_i$ be equal to $\{(p, q) \mid \langle p, \varepsilon \rangle \xrightarrow{u_i} \langle q, \varepsilon \rangle\}$ for all $i$. Then after reading $w$, the det-VPA $\mathcal{B}$ has its current state equal to $R_{n+1}$ and its stack containing $(R_n, a_n) \ldots (R_2, a_2)(R_1, a_1)$. Assume we are reading the symbol $a$ after $w$, then $\mathcal{B}$ performs the following transition from $R_{n+1}$: (1) if $a \in \Sigma_c$, then push $(R_{n+1}, a)$ on the stack and go to the state $R = \mathbb{I}_Q$ (a new unmatched call symbol is read); (2) if $a \in \Sigma_i$, then go to the state $R = \{(p, q) \mid \exists (p, p') \in R_{n+1}, (p', a, q) \in \delta_i\}$ ($u_{n+1}$ is extended to the well-matched word $u_{n+1}a$); (3) if $a \in \Sigma_r$, then pop $(R_n, a_n)$ from the stack if $\bar{a}_n = a$, and go to the state

$$R = \{(p, q) \mid \exists (p, p') \in R_n, (p', a_n, r', \gamma) \in \delta_c, (r', r) \in R_{n+1}, (r, a, \gamma, q) \in \delta_r\}$$

[4] The stack symbol $\gamma$ is pushed on the left of $\sigma$.

[5] The original definition of VPA [3] allows acceptance of ill-matched words.

(the call symbol $a_n$ is matched with the return symbol $a = \bar{a}_n$, leading to the well-matched word $u_n a_n u_{n+1} a$). Finally the initial state of $\mathcal{B}$ is $\mathbb{I}_{Q_I}$ and its final states are sets $R$ containing some $(p, q)$ with $p \in Q_I$ and $q \in Q_F$. □

Though a VPL $L$ in general does not have a unique minimal det-VPA $\mathcal{A}$ accepting $L$, imposing the following subclass leads to a unique minimal acceptor.

**Definition 2 ( [2, 9]).** *A* 1-module single entry VPA[6] *(1-SEVPA) is a det-VPA $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I = \{q_0\}, Q_F)$ such that its stack alphabet $\Gamma$ is equal to $Q \times \Sigma_c$, and all its call transitions $(q, a, q', \gamma) \in \delta_c$ are such that $q' = q_0$ and $\gamma = (q, a)$.*

**Theorem 2 ( [2]).** *For any VPL $L$, there exists a unique minimal (with regards to the number of states) 1-SEVPA accepting $L$, up to a renaming of the states.*[7]

**Learning VPAs** Let us recall the concept of *learning* a deterministic finite automaton (DFA), as introduced in [4]. Let $L$ be a regular language over an alphabet $\Sigma$. The task of the *learner* is to construct a DFA $\mathcal{H}$ such that $\mathcal{L}(\mathcal{H}) = L$ by interacting with the *teacher*. The two possible types of interactions are *membership queries* (does $w \in \Sigma^*$ belong to $L$?), and *equivalence queries* (does the DFA $\mathcal{H}$ accept $L$?). For the latter type, if the answer is negative, the teacher also provides a counterexample, i.e., a word $w$ such that $w \in L \Leftrightarrow w \notin \mathcal{L}(\mathcal{H})$. The so-called $L^*$ *algorithm* of [4] learns at least one representative per equivalence class of the Myhill-Nerode congruence of $L$ [8] from which the minimal DFA $\mathcal{D}$ accepting $L$ is constructed. This learning process terminates and it uses a polynomial number of membership and equivalence queries in the size of $\mathcal{D}$, and in the length of the longest counterexample returned by the teacher [4].

In [9], an extension of Angluin's learning algorithm is given for VPLs. The Myhill-Nerode congruence for regular languages is extended to VPLs as follows. Given a pushdown alphabet $\tilde{\Sigma}$ and a VPL $L$ over $\tilde{\Sigma}$, we consider the set of *context pairs*$\mathrm{CP}(\tilde{\Sigma}) = \{(u, v) \in (\mathrm{WM}(\tilde{\Sigma}) \cdot \Sigma_c)^* \times \mathrm{Suff}(\mathrm{WM}(\tilde{\Sigma})) \mid \beta(u) = -\beta(v)\}$, and we define the equivalence relation $\simeq_L \subseteq \mathrm{WM}(\tilde{\Sigma}) \times \mathrm{WM}(\tilde{\Sigma})$ [2, 9] such that $w \simeq_L w'$ if and only if $\forall (u, v) \in \mathrm{CP}(\tilde{\Sigma}), uwv \in L \Leftrightarrow uw'v \in L$. The minimal 1-SEVPA accepting $L$ as described in Theorem 2 is constructed from $\simeq_L$ such that its states are the equivalence classes of $\simeq_L$.

**Theorem 3 ( [9]).** *Let $L$ be a VPL over $\tilde{\Sigma}$ and $n$ be the index of $\simeq_L$. queries and a number of membership queries polynomial in $n$, $|\Sigma|$, and $\log \ell$, where $\ell$ is the length of the longest counterexample returned by the teacher.*

The learning process designed in [9] extends to VPLs the TTT *algorithm* proposed in [10] for regular languages. TTT improves the efficiency of the $L^*$ algorithm by eliminating redundancies in counterexamples provided by the teacher.

[6] The definitions of 1-SEVPA in [2] and [9] differ slightly. We follow the one in [9].

[7] This 1-SEVPA may be exponentially bigger than the size of a VPA accepting $L$.

## 3 JSON Format

In this section, we describe JSON documents [6] and JSON schemas [13] that impose some constraints on the structure of JSON documents. We also present the abstractions we make for the purpose of this paper.

**JSON Documents** We describe the structure of JSON documents. Our presentation is inspired by [19], though some details are skipped for readability (see [14] for a full description). The JSON format defines different types of *JSON values*:

- `true`, `false`, `null` are JSON values. Any decimal number (positive, negative) is a JSON value, called a *number*. In particular any number that is an integer is called an *integer*. Any finite sequence of characters starting and ending with `"` is a *string value*. All those values are called *primitive values*.
- If $v_1, v_2, \ldots, v_n$ are JSON values and $k_1, k_2, \ldots, k_n$ are *pairwise distinct* string values, then $\{k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n\}$ is a JSON value, called an *object*. Each $k_i : v_i$ is called a *key-value pair* such that $k_i$ is the *key*. The collection of these pairs is *unordered*.
- If $v_1, v_2, \ldots, v_n$ are JSON values, then $[v_1, v_2, \ldots, v_n]$ is a JSON value, called an *array*. Each $v_i$ is an *element* and the collection thereof is *ordered*.

In this work, *JSON documents* are supposed to be objects.[8] One can use *JSON pointers* to navigate through a document, e.g., if $J$ is an object and $k$ is a key, then $J[k]$ is the value $v$ such that the key-value pair $k{:}v$ appears in $J$.

In this paper, we consider somewhat *abstract* JSON documents. We see JSON documents as well-matched words over the pushdown alphabet $\tilde{\Sigma}_{\mathrm{JSON}}$ that we describe hereafter. We abstract all string values as `s`, and all numbers as `n` (as `i` when they are integers). We denote by $\Sigma_{\mathrm{pVal}} = \{\mathtt{true}, \mathtt{false}, \mathtt{null}, \mathtt{s}, \mathtt{n}, \mathtt{i}\}$ the alphabet composed of the six primitive values. Concerning the key-value pairs appearing in objects, each key together with the symbol ":" following the key is abstracted as an alphabet symbol $k$. We assume knowledge of a *finite* alphabet $\Sigma_{\mathrm{key}}$ of keys. We define the pushdown alphabet $\tilde{\Sigma}_{\mathrm{JSON}} = (\Sigma_c, \Sigma_r, \Sigma_i)$ with $\Sigma_i = \Sigma_{\mathrm{key}} \cup \Sigma_{\mathrm{pVal}} \cup \{\#\}$, where $\#$ is used in place of the comma; $\Sigma_c = \{\prec, \sqsubset\}$, where $\prec$ (resp. $\sqsubset$) is used in place of "{" (resp. "["); and $\Sigma_r = \{\succ, \sqsupset\}$, with $\overline{\prec} = \succ$ and $\overline{\sqsubset} = \sqsupset$. We denote by $\Sigma_{\mathrm{JSON}}$ the set $\Sigma_c \cup \Sigma_r \cup \Sigma_i$.

*Example 1.* An example of a JSON document is given in Listing 1. We can see that this document is an object containing three keys: `"title"`, whose associated value is a string value; `"keywords"`, whose value is an array containing string values; and `"conf"`, whose value is an object. This inner object contains two keys: `"name"`, whose value is a string value; `"year"`, whose value is an integer. The pointer `J[conf][name]`, where `J` is the root of the document, retrieves the value `"TACAS"`. The JSON document is abstracted as the word $\prec k_1 \mathtt{s} \# k_2 \sqsubset \mathtt{s} \# \mathtt{s} \# \mathtt{s} \sqsupset \# k_3 \prec k_4 \mathtt{s} \# k_5 \mathtt{i} \succ \succ \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$ where $\Sigma_{\mathrm{key}}$ contains the keys $k_i, i \in \{1, \ldots, 5\}$.

[8] In [6], a JSON document can be any JSON value and duplicated keys are allowed inside objects. In this paper, we follow what is commonly used in practice: JSON documents are objects, and keys are pairwise distinct inside objects.

```
{ "title": "Validating Streaming JSON Documents with Learned VPAs",
  "keywords": ["VPA", "JSON documents", "streaming validation"],
  "conf": { "name": "TACAS", "year": 2023 }
}
```

Listing 1: A JSON document.

```
{ "type": "object",
  "required": ["title", "conf"],
  "properties": {
    "title": { "type": "string" },
    "keywords": { "type": "array", "items": { "type": "string" } },
    "conf": {
      "type": "object",
      "required": ["name", "year"],
      "properties":{ "name":{"type": "string"},"year":{"type": "integer"}}}}}
```

Listing 2: A JSON schema.

**JSON Schemas** A *JSON schema* can impose some constraints on JSON documents by specifying any of the types of JSON values that appear in those documents. We say that a JSON document *satisfies* (or is *valid* for) the schema if it verifies the constraints imposed by this schema. We denote by $\mathcal{L}(S)$ the set of documents that are valid for $S$. In this section, we give a simplified presentation of JSON schemas and refer to [13] for a complete description and to [19] for a formalization (i.e. a formal grammar with its syntax and semantics).

A JSON schema is itself a JSON document that uses several keywords that help shape and restrict the set of JSON documents that this schema specifies. As we abstract JSON documents, JSON schemas we work on are also abstracted. We do not consider the restrictions that can be imposed on string values and numbers, for instance. We give here a few examples. See [13] for more details.

- Within object schemas, restrictions can be imposed on the key-value pairs of the objects, e.g., the value associated with some key has itself to satisfy a certain schema, or some particular keys must be present in the object.
- Within array schemas, it can be imposed that all elements of the array satisfy a certain schema, or that the array has a minimum/maximum size.
- Schemas can be combined with Boolean operations, e.g., a JSON document must satisfy a conjunction of several schemas.
- A schema can be defined as one referred to by a JSON pointer. This allows a *recursive* structure for the JSON documents satisfying a certain schema.

*Example 2.* The schema from Listing 2 describes objects that can have three keys: `"title"`, whose associated value must be a string value; `"keywords"`, an array of strings; and `"conf"`, an object. Among these, `"title"` and `"conf"` are required. The JSON document of Example 1 satisfies this JSON schema.

Under these abstractions, we can always construct a VPA that accept the same set of JSON documents than a schema $S$, as shown in the following theorem. We also extend this construction to the case where we fix an *order* $<$

on $\Sigma_{\text{key}}$ and consider the set $\mathcal{L}_<(S)$ of documents valid for $S$ whose key order inside objects respects this order $<$. The main idea of the proof is to define a formalism of JSON schemas as *extended context-free grammars*, and show that we can construct a VPA from such a grammar.

**Theorem 4.** *Let $S$ be a JSON schema. Then, there exists a VPA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A})$ is the set $\mathcal{L}(S)$ of documents valid with regards to $S$. Moreover, for any order $<$ of $\Sigma_{\text{key}}$, there exists a VPA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}_<(S)$.*

Our proof does not give a construction of the grammar from the schema $S$. The grammar depends on the formal semantics of JSON schemas which are still changing and being debated. Thus, to be more robust to changes in the semantics, we prefer to learn the minimal 1-SEVPA $\mathcal{B}$ accepting $\mathcal{L}_<(S)$ given a fixed order $<$, in the sense of Theorem 3.[9] For learning, equivalence queries require to generate a certain number of random JSON documents.[10] If $S$ and the learner's hypothesis $\mathcal{H}$ disagree on a document, we have a counterexample. Otherwise, we say that $\mathcal{H}$ is correct. In both membership and equivalence queries, we only accept documents whose key order inside objects satisfy the order $<$. The randomness used in the equivalence queries implies that the learned 1-SEVPA may not exactly accept $\mathcal{L}_<(S)$. Setting the number of generated documents to be large would help reducing the probability that an incorrect 1-SEVPA is learned.

## 4 Validation of JSON Documents

For this section, let us fix a schema $S$, an order $<$ on $\Sigma_{\text{key}}$, and a 1-SEVPA $\mathcal{A} = (Q, \tilde{\Sigma}_{\text{JSON}}, \Gamma, \delta, \{q_0\}, Q_F)$ accepting $\mathcal{L}_<(S)$. We present a *streaming* algorithm to decide if a document $J$ is in $\mathcal{L}(S)$. By "streaming", we mean an algorithm that processes the document in a single pass, symbol by symbol. Our new approach is as follows. We learn $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$. As $\mathcal{L}_<(S) \neq \mathcal{L}(S)$, we design an algorithm that uses $\mathcal{A}$ in a clever way to allow arbitrary key orders in documents to validate. To do this, we use a *key graph* defined in the sequel.

**Key Graph** In this section, w.l.o.g. we suppose that $\mathcal{A}$ has *no bin states*. Let $\mathrm{T}_{\mathcal{A}}$ be the transition system of $\mathcal{A}$. We explain how to associate to $\mathcal{A}$ its *key graph* $\mathrm{G}_{\mathcal{A}}$: an abstraction of the paths of $\mathrm{T}_{\mathcal{A}}$ labeled by the contents of the objects appearing in words of $\mathcal{L}_<(S)$. This graph is essential in our validation algorithm.

**Definition 3.** *The* key graph $\mathrm{G}_{\mathcal{A}}$ *of $\mathcal{A}$ has:*

- *the vertices $(p, k, p')$ with $p, p' \in Q$ and $k \in \Sigma_{\text{key}}$ if there exists in $\mathrm{T}_{\mathcal{A}}$ a path $\langle p, \varepsilon\rangle \xrightarrow{kv} \langle p', \varepsilon\rangle$ with $v \in \Sigma_{\text{pVal}} \cup \{au\bar{a} \mid a \in \Sigma_c, u \in \mathrm{WM}(\tilde{\Sigma}_{\text{JSON}})\}$,*[11]

[9] We use this automaton in the next section for the validation of JSON documents. We do not use a 1-SEVPA for $\mathcal{L}(S)$ as it could be exponentially larger.

[10] It is common to proceed this way in automata learning, as explained in [4, Sec. 4].

[11] Notice that each vertex $(p, k, p')$ of $\mathrm{G}_{\mathcal{A}}$ only stores the key $k$ and not the word $kv$.

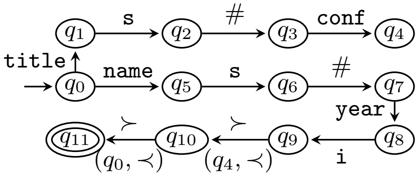


Fig. 1: A 1-SEVPA for the schema from Listing 2, without the key keywords.

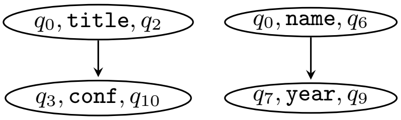


Fig. 2: The key graph for the 1-SEVPA from Figure 1.

- *the edges* $((p_1, k_1, p'_1), (p_2, k_2, p'_2))$ *if there exists* $(p'_1, \#, p_2) \in \delta_i$.

We have the following property.

**Lemma 1.** *There exists a path* $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ *in* $\mathrm{G}_{\mathcal{A}}$ *with* $p_1 = q_0$ *if and only if there exist a factor* $u$ *of a word in* $\mathcal{L}_<(S)$ *such that* $u = k_1 v_1 \# \dots \# k_n v_n$ *where each* $k_i v_i$ *is a key-value pair, and a path* $\langle q_0, \varepsilon \rangle \xrightarrow{u} \langle p'_n, \varepsilon \rangle$ *in* $\mathrm{T}_{\mathcal{A}}$ *that decomposes as* $\langle p_i, \varepsilon \rangle \xrightarrow{k_i v_i} \langle p'_i, \varepsilon \rangle, \forall i \in \{1, \dots, n\}$ *and* $\langle p'_i, \varepsilon \rangle \xrightarrow{\#} \langle p_{i+1}, \varepsilon \rangle, \forall i \in \{1, \dots, n-1\}$. *Furthermore, there is no path* $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ *such that* $k_i = k_j$ *for some* $i \neq j$. *That is,* $\mathrm{G}_{\mathcal{A}}$ *contains a finite number of paths.*

Hence, paths in $\mathrm{G}_{\mathcal{A}}$ focus on contents of objects being part of JSON documents in $\mathcal{L}_<(S)$. Moreover, they abstract paths in $\mathrm{T}_{\mathcal{A}}$ in the sense that only keys $k_i$ are stored and the subpaths labeled by the values $v_i$ are implicit.

*Example 3.* Consider the schema from Listing 2, without the key keywords. A 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(S)$ is given in Figure 1. For clarity, call transitions[12] and the bin state are not represented. In Figure 2, we depict its corresponding key graph $\mathrm{G}_{\mathcal{A}}$. Since we have the path $\langle q_0, \varepsilon \rangle \xrightarrow{\texttt{title s}} \langle q_2, \varepsilon \rangle$ in $\mathrm{T}_{\mathcal{A}}$, the triplet $(q_0, \texttt{title}, q_2)$ is a vertex of $\mathrm{G}_{\mathcal{A}}$. Likewise, $(q_0, \texttt{name}, q_6)$ and $(q_7, \texttt{year}, q_9)$ are vertices. As we have the path $\langle q_4, \varepsilon \rangle \xrightarrow{\prec} \langle q_0, (q_4, \prec) \rangle \xrightarrow{\texttt{name s \# year i}} \langle q_9, (q_4, \prec) \rangle \xrightarrow{\succ} \langle q_{10}, \varepsilon \rangle$, $(q_3, \texttt{conf}, q_{10})$ is also a vertex of $\mathrm{G}_{\mathcal{A}}$. Finally, as $\langle q_2, \varepsilon \rangle \xrightarrow{\#} \langle q_3, \varepsilon \rangle$, we have an edge from $(q_0, \texttt{title}, q_2)$ to $(q_3, \texttt{conf}, q_{10})$.

Computing the key graph can be done in polynomial time by first computing the reachability relation $\mathrm{Reach}_{\mathcal{A}}$. From this relation, the vertices can be easily found. Since the edges require to check whether a transition reading # exists, it is obvious that it can be done in polynomial time.

**Validation Algorithm** In this section, we provide a streaming algorithm that validates JSON documents against a given JSON schema $S$.

Given a word $w \in \Sigma^*_{\mathrm{JSON}} \setminus \{\varepsilon\}$, we want to check whether $w \in \mathcal{L}(S)$. The main difficulty is that the key-value pairs inside an object are arbitrarily ordered in $w$ while a fixed key order $<$ is encoded in the 1-SEVPA $\mathcal{A}$ ($\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$).

[12] Recall the form of call transitions for 1-SEVPAs, see Definition 2.

Our validation algorithm is inspired by the algorithm computing a det-VPA equivalent to some given VPA [3] (see Theorem 1 and its proof) and uses the key graph $\mathrm{G}_{\mathcal{A}}$ to treat arbitrary orders of the key-value pairs inside objects.

During the reading of $w \in \Sigma_{\mathrm{JSON}}^{*} \setminus \{\varepsilon\}$, in addition to checking whether $w \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$, the algorithm updates a subset $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and modifies the content of a stack $\mathit{Stk}$ (push, pop, modify the element on top of $\mathit{Stk}$).

First, let us explain the information stored in $R$. Assume that we have read the prefix $zau$ of $w$ such that $a \in \Sigma_c$ is the last unmatched call symbol (thus $za \in (\mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}}) \cdot \Sigma_c)^*$ and $u \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$).

- If $a$ is the symbol $\sqsubset$, then we have $R = \{(p, q) \mid \langle p, \varepsilon\rangle \xrightarrow{u} \langle q, \varepsilon\rangle\}$.
- If $a$ is the symbol $\prec$, then we have $u = k_1 v_1 \# k_2 v_2 \# \ldots k_{n-1} v_{n-1} \# u'$ such that $u' \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$ and $u'$ is prefix of $k_n v_n$, where each $k_i v_i$ is a key-value pair. Then $R = \{(p, q) \mid \langle p, \varepsilon\rangle \xrightarrow{u'} \langle q, \varepsilon\rangle\}$.

In the first case, by using $R$ as defined previously, we adopt the same approach as for the determinization of VPAs. In the second case, with $u$, we are currently reading the key-value pairs of an object in some order, not necessarily the one encoded in $\mathcal{A}$. In this case the set $R$ is focused on the currently read key-value pair $k_n v_n$, that is, on the word $u'$. After reading of the whole object $\prec k_1 v_1 \# k_2 v_2 \# \ldots \succ$, we will use the key graph $\mathrm{G}_{\mathcal{A}}$ to update the current set $R$.

Second, an element stored in the stack $\mathit{Stk}$ is either a pair $(R, \sqsubset)$, or a 5-tuple $(R, \prec, K, k, \mathit{Bad})$, where $R$ is a set as described previously, $K \subseteq \Sigma_{\mathrm{key}}$ is a subset of keys, $k \in \Sigma_{\mathrm{key}}$ is a key, and $\mathit{Bad}$ is a set containing some vertices of $\mathrm{G}_{\mathcal{A}}$.[13]

We now detail our streaming validation algorithm.[14] Before reading $w$, we initialize $R$ to the set $\mathbb{I}_{\{q_0\}}$ and $\mathit{Stk}$ to the empty stack. Let us explain how to update the current set $R$ and the current content of the stack $\mathit{Stk}$ while reading the input word $w$. Suppose that we are reading the symbol $a$ in $w$. In some cases we will also peek the symbol $b$ following $a$ (*lookahead* of one symbol).

**Case (1)** Suppose that $a$ is the symbol $\sqsubset$, i.e., we start an array. Hence $(R, \sqsubset)$ is pushed on $\mathit{Stk}$ and $R$ is updated to $R_{\mathit{Upd}} = \mathbb{I}_{\{q_0\}}$. We thus proceed as in the proof of Theorem 1 (with $\mathbb{I}_{\{q_0\}}$ instead of $\mathbb{I}_Q$, since $\mathcal{A}$ is a 1-SEVPA[12]).
**Case (2)** Suppose that $a \in \Sigma_i$ and $\sqsubset$ appears on top of $\mathit{Stk}$. We are thus reading the elements of an array. Hence $R$ is updated to $R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, q') \in R, (q', a, q) \in \delta_i\}$. Again we proceed as in the proof of Theorem 1.
**Case (3)** Suppose that $a$ is the symbol $\sqsupset$. This means that we finished reading an array. If the stack is empty or its top element contains $\prec$, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Otherwise $(R', \sqsubset)$ is popped from $\mathit{Stk}$ and $R$ is updated to $R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', (p', \sqsubset, q_0, \gamma) \in \delta_c, (q_0, r) \in R, (r, \sqsupset, \gamma, q) \in \delta_r\}$, as in the proof of Theorem 1.
**Case (4)** Suppose that $a$ is the symbol $\prec$.

[13] In the particular case of the object $\prec\succ$, the 5-tuple $(R, \prec, K, k, \mathit{Bad})$ is replaced by $(R, \prec)$. This situation will be clarified during the presentation of our algorithm.

[14] Note that the algorithm assumes we have a 1-SEVPA.

– Let us first consider the particular case where the symbol $b$ following $\prec$ is equal to $\succ$, meaning that we will read the object $\prec\succ$. In this case, $(R, \prec)$ is pushed on $\mathit{Stk}$ and $R$ is updated to $R_{\mathit{Upd}} = \mathbb{I}_{\{q_0\}}$ as in Case (1).

– Otherwise, if $b$ belongs to $\Sigma_{\text{key}}$, we begin to read a (non-empty) object whose treatment is different from that of an array as its key-value pairs can be read in any order. Then, $R$ is updated to $R_{\mathit{Upd}} = \mathbb{I}_{P_b}$ where $P_b = \{p \in Q \mid \exists (p, b, p') \in \mathrm{G}_{\mathcal{A}}\}$, and $(R, \prec, K, b, \mathit{Bad})$ is pushed on $\mathit{Stk}$ such that $K$ is the singleton $\{b\}$ and $\mathit{Bad}$ is the empty set. The 5-tuple pushed on $\mathit{Stk}$ indicates that the key-value pair that will be read next begins with key $b$; moreover $K = \{b\}$ because this is the first pair of the object. The meaning of $\mathit{Bad}$ will be clarified later. The updated set $R_{\mathit{Upd}}$ is equal to the identity relation on $P_b$ since after reading $\prec$, we will start reading a key-value pair whose abstracted state in $\mathrm{G}_{\mathcal{A}}$ can be any state from $P_b$. Later while reading the object whose reading is here started, we will update the 5-tuple on top of $\mathit{Stk}$ as explained below.

– Finally, it remains to consider the case where $b \notin \Sigma_{\text{key}} \cup \{\succ\}$. In this final case, we have that $w \notin \mathcal{L}(S)$ and we stop the algorithm.

**Case (5)** Suppose that $a \in \Sigma_i \setminus \{\#\}$ and $\prec$ appears on top of $\mathit{Stk}$. Therefore, we are currently reading a key-value pair of an object. Then $R$ is updated to $R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, q') \in R, (q', a, q) \in \delta_i\}$.

**Case (6)** Suppose that $a$ is the symbol $\#$ and $\prec$ appears on top of $\mathit{Stk}$. This means that we just finished reading a key-value pair whose key $k$ is stored in the 5-tuple $(R', \prec, K, k, \mathit{Bad})$ on top of $\mathit{Stk}$, and that another key-value pair will be read after symbol $\#$. The set $K$ in $(R', \prec, K, k, \mathit{Bad})$ stores all the keys of the key-values pairs already read including $k$.

– If the symbol $b$ following $\#$ does not belong to $\Sigma_{\text{key}}$, then $w \notin \mathcal{L}(S)$ and we stop the algorithm.

– Otherwise, if $b$ belongs to $K$, this means that the object contains twice the same key, that is, $w \notin \mathcal{L}(S)$, and we also stop the algorithm.

– Otherwise, the set $R$ is updated to $R_{\mathit{Upd}} = \mathbb{I}_{P_b}$ (as we begin the reading of a new key-value pair whose key is $b$) and the 5-tuple $(R', \prec, K, k, \mathit{Bad})$ on top of $\mathit{Stk}$ is updated such that *(i)* $K$ is replaced by $K \cup \{b\}$, *(ii)* $k$ is replaced by $b$, and *(iii)* all vertices $(p, k, p')$ of $\mathrm{G}_{\mathcal{A}}$ such that $(p, p') \notin R$ are added to the set $\mathit{Bad}$. Recall that the vertex $(p, k, p')$ of $\mathrm{G}_{\mathcal{A}}$ is a witness of a path $\langle p, \varepsilon\rangle \xrightarrow{kv} \langle p', \varepsilon\rangle$ in $\mathrm{T}_{\mathcal{A}}$ for some key-value pair $kv$. Hence by adding this vertex $(p, k, p')$ to $\mathit{Bad}$, we mean that the pair that has just been read does not use such a path.

**Case (7)** Suppose that $a$ is the symbol $\succ$. Therefore we end the reading of an object. If the stack is empty or its top element contains $\sqsubset$, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Otherwise the top of $\mathit{Stk}$ contains either $(R', \prec)$ or $(R', \prec, K, k, \mathit{Bad})$ that we pop from $\mathit{Stk}$.

– If $(R', \prec)$ is popped, then we are ending the reading of the object $\prec\succ$. Hence, we proceed as in Case (3): $R$ is updated to $R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', (p', \prec, q_0, \gamma) \in \delta_c, (q_0, \succ, \gamma, q) \in \delta_r\}$.[15]

[15] Notice that $R$ does not appear in $R_{\mathit{Upd}}$ as $R = \mathbb{I}_{\{q_0\}}$.

– If $(R', \prec, K, k, \mathit{Bad})$ is popped, we are ending an object whose last seen key is $k$. As in Case (6), we add to $\mathit{Bad}$ all vertices $(p, k, p')$ such that $(p, p') \notin R$. Let $\text{Valid}(K, \mathit{Bad})$ be the set of pairs of states $(q_0, r')$ such that there exists a path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ in $\mathrm{G}_\mathcal{A}$ with $p_1 = q_0$, $p'_n = r'$, $(p_i, k_i, p'_i) \notin \mathit{Bad}$ for all $i \in \{1, \dots, n\}$, and $K = \{k_1, \dots, k_n\}$. Then $R$ is updated to $R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', (p', \prec, q_0, \gamma) \in \delta_c, (q_0, r) \in \text{Valid}(K, \mathit{Bad}), (r, \succ, \gamma, q) \in \delta_r\}$. We thus proceed as in Case (3) except that condition $(r', r) \in R$ is replaced by $(r', r) \in \text{Valid}(K, \mathit{Bad})$. That way, we check that the key-value pairs that have been read as composing an object of $w$ label some path in $\mathrm{T}_\mathcal{A}$, once ordered by $<$. That is, the corresponding abstract path appears in $\mathrm{G}_\mathcal{A}$.

**Case (8)** Suppose that $a \in \Sigma_i$ and $\mathit{Stk}$ is empty, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Indeed an internal symbol appears either in an array or in an object (see Cases (2), (5), and (6) above).

Finally, when the input word $w$ is completely read, we check whether the stack $\mathit{Stk}$ is empty and the computed set $R$ contains a pair $(q_0, q)$ with $q \in Q_F$.

The complexity of our algorithm is given in the following proposition.

**Proposition 1.** *Let $S$ be a schema and $\mathcal{A}$ be a 1-SEVPA such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$. Deciding if a document $J$ is valid is in time $\mathcal{O}(|J| \cdot (|Q|^4 + |Q|^{|\Sigma_{\text{key}}|} \cdot |\Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|+1}))$, and uses $\mathcal{O}(|\delta| + |Q|^2 \cdot |\Sigma_{\text{key}}| + d(J) \cdot (|Q|^2 + |\Sigma_{\text{key}}|))$ memory.*

## 5 Implementation and Experiments

We present here our Java implementation of the learning process and the validation algorithm. First, we present classical validation algorithms and explain how to generate documents from a schema. We then explain how the required membership and equivalence queries are implemented. Finally, we present the schemas we evaluated, and the results for the learning, computation of the key graph, and validation experiments. The reader is referred to the code documentation for more details about our implementation [27–31].

In the remaining of this section, let us assume we have a JSON schema $S_0$.

**Classical Validation Algorithm and Documents Generation** Let us explain briefly the *classical* algorithm used in many implementations for validating a JSON document $J_0$ against $S_0$ [13]. It is a recursive algorithm that follows the constraints of $S_0$.[16] For instance, if the current value $J$ is an object, we iterate over each key-value pair in $J$ and its corresponding sub-schema in the current schema $S$. Then, $J$ satisfies $S$ if and only if the values in the key-value pairs all satisfy their corresponding sub-schema. As long as $S_0$ does not contain any Boolean operations, this algorithm is straightforward and linear in the size of both the initial document $J_0$ and schema $S_0$. However, if $S_0$ contains Boolean operations, then the current value $J$ may be processed multiple times.

[16] Such a recursive algorithm is briefly presented in [19].

In order to match the abstractions we defined (see Section 3) and to have options to tune the learning process, we implemented our own classical validator. Alongside the validator, we implemented a tool to generate JSON documents whose structure is dictated by $S_0$. Due to the Boolean operations $S_0$ can contain, it may happen that choices must be made during the generation process. We have two generators: a *random* generator that makes a choice at random, and an *exhaustive* generator that exhaustively explores every choice, thus producing every valid document one by one. Moreover, we implemented modifications of these generators to allow the creation of invalid documents, by allowing *deviations*.[17] For instance, if the current schema describes an integer, we can instead decide to generate a string. To ensure we eventually produce a document, we can fix a *maximal depth* (i.e., the maximal number of nested objects or arrays). This is useful for recursive schemas, or when generating invalid documents.

**Learning Algorithm** Let us now focus on the learning algorithm itself, and in particular on the membership and equivalence queries. We recall that the equivalence queries are performed by generating a certain number of (valid and invalid) JSON documents and by verifying that the learned VPA $\mathcal{H}$ and the given schema $S_0$ agree on the documents' validity. As said in Section 2, we use the TTT algorithm [9] to learn a 1-SEVPA from $S_0$, relying on its implementation in the well-known Java libraries LearnLib and AutomataLib [11].

We use the random and exhaustive generators of valid and invalid documents as explained above and we fix two constants $C$ and $D$ depending on the schema to be learned.[18] For a *membership* query over a word $w \in \Sigma_{\text{JSON}}^*$, the teacher runs the classical validator on $w$ and $S_0$. For an *equivalence* query over a learned 1-SEVPA $\mathcal{H}$, the teacher uses a generator to produce documents on which $\mathcal{H}$ is tested. If that generator is random, at each query, $C$ documents are generated for each document depth between 0 and $D$. If none of the documents leads to a counterexample, the teacher checks whether $\mathrm{G}_{\mathcal{H}}$ does not satisfy Lemma 1, i.e., whether there is path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ with $p_1 = q_0$ such that $k_i = k_j$ for some $i \neq j$. In that case, we can create a counterexample.

**Evaluated Schemas** For the experimental evaluation of our algorithms, we consider the following schemas, sorted in increasing size: (1) A schema that accepts documents defined recursively. Each object contains a string and can contain an array whose single element satisfies the whole schema, i.e., this is a recursive list. (2) A schema that accepts documents containing each type of values, i.e., an object, an array, a string, a number, an integer, and a Boolean. (3) A schema that defines how snippets must be described in *Visual Studio Code* [23]. (4) A recursive schema that defines how the metadata files for *VIM plugins* must be written [22]. (5) A schema that defines how *Azure Functions Proxies* files must look like [20]. (6) A schema that defines the configuration file

[17] This is similar to mutation testing [1,12].

[18] The values of $C$ and $D$ are given below.

for a code coverage tool called *codecov* [21]. Hence, we consider two schemas written by ourselves to test our framework, and four schemas that are used in real world cases. The last four schemas were modified to make all object keys mandatory and to remove unsupported keywords. All used schemas and scripts can be consulted on our repository [30]. In the rest of this section, the schemas are referred to by their order in the previous enumeration.

We present three types of experimental results: (1) the time and number of membership and equivalence queries to learn a 1-SEVPA $\mathcal{A}$ from a JSON schema, (2) the time and memory to compute the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ and the key graph $\mathrm{G}_{\mathcal{A}}$, and (3) the time and memory to validate a document using both classical and new algorithms. The server used for the benchmarks ran OpenJDK version 11.0.12 on Debian 10 over Linux 5.4.73-1-pve with a 4-core Intel® Xeon® Silver 4214R Processor with 16.5M cache, and 64GB of RAM.

**Learning VPAs** First, we learn a 1-SEVPA from a schema. We use an exhaustive generator for the first three schemas (accepting a small set of documents), and a random generator[19] for the remaining three for which we fix $C = 10000$. For both generators, we set $D = \mathit{depth}(S) + 1$, where $\mathit{depth}(S)$ is the maximal number of nested objects and arrays in the schema $S$, except for the recursive list where $D = 10$, and for the recursive *VIM plugin* schema where $D = 7$.

For the first five schemas, we do not set a time limit and repeat the learning process ten times. For the last schema, we set a time limit of one week and, for time constraints, only perform the learning process once. After that, we stop the computation and retrieve the learned 1-SEVPA at that point. The retrieved automaton is therefore an approximation of this schema. Its key graph has repeated keys along some of its paths, a situation that cannot occur if the 1-SEVPA was correctly learned, see Lemma 1. Results are given in Table 1.

**Comparing Validation Algorithms** The second part of the preprocessing step is to construct the key graph of the learned 1-SEVPA. For each evaluated schema, we select the learned automaton with the largest set of states, in order to report a worst-case measure. Results after a single experiment are given in Table 2. We can see that the storage of the key graph does not consume more than one megabyte, except for *codecov* schema. That is, even for non-trivial schemas, the key graph is relatively lightweight.

Finally, we compare both classical and new streaming validation algorithms. For the latter, we use the 1-SEVPA (and its key graph) selected as described above. We first generate 5000 valid and 5000 invalid JSON documents using a random generator, with a maximal depth equal to $D = 20$. We then measure the time and memory required by both validation algorithms on these documents.[20]

[19] With the random generator, the learned 1-SEVPAS may differ each experiment.

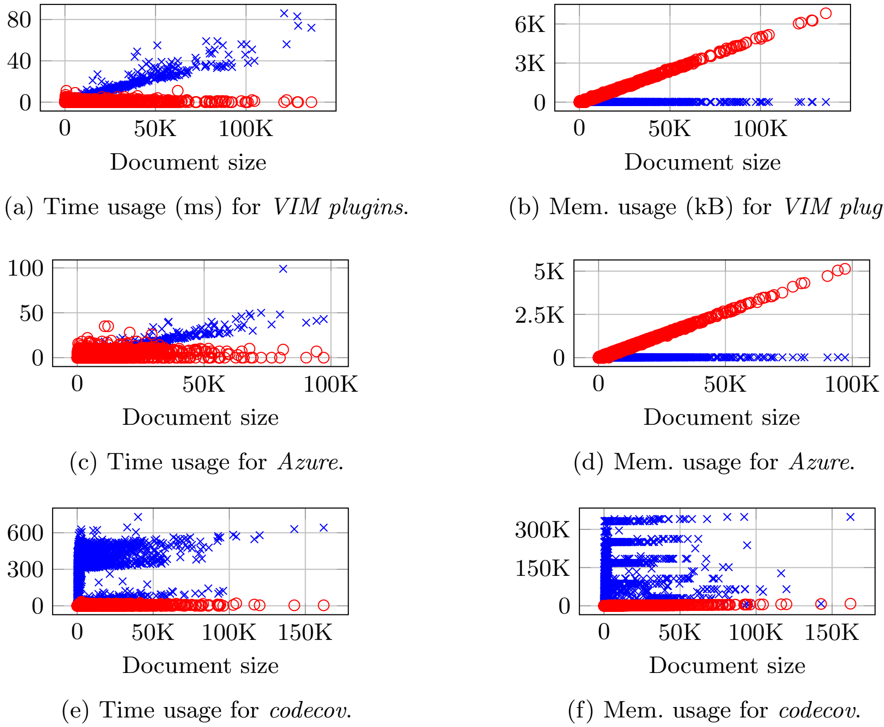
[20] Since obtaining a close approximation of the consumed memory requires Java to stop the execution and destroy all unused objects, we execute each algorithm twice: once to measure time, and a second time to measure memory.

| Time (s) | Membership | Equivalence | $\lvert Q\rvert$ | $\lvert\Sigma\rvert$ | $\lvert\delta_c\rvert$ | $\lvert\delta_r\rvert$ | $\lvert\delta_i\rvert$ | Diameter |
|---|---|---|---|---|---|---|---|---|
| 2.2 | 2055.0 | 5.0 | 7.0 | 15.0 | 14.0 | 3.0 | 5.0 | 3.0 |
| 4.5 | 69514.0 | 3.0 | 24.0 | 20.0 | 48.0 | 3.0 | 26.0 | 12.0 |
| 9.0 | 21943.0 | 5.0 | 16.0 | 17.0 | 32.0 | 7.0 | 18.0 | 13.0 |
| 9590.3 | 4246085.0 | 36.4 | 150.0 | 27.0 | 300.0 | 2946.5 | 760.3 | 9.0 |
| 35008.2 | 4063971.7 | 30.5 | 121.0 | 35.0 | 242.0 | 2123.0 | 752.5 | 13.3 |
| Timeout | 633049534.0 | 192.0 | 884.0 | 77.0 | 1768.0 | 89695.0 | 8557.0 | 28.0 |

Table 1: Learning results. For the first five schemas, values are averaged out of ten experiments. For the last schema, a single experiment was conducted.

| Reach$_{\mathcal{A}}$ | | | G$_{\mathcal{A}}$ | | | |
|---|---|---|---|---|---|---|
| Time (s) | Memory (kB) | Size | Time (s) | Computation (kB) | Storage (kB) | Size |
| 34 | 492 | 31 | 100 | 2231 | 65 | 3 |
| 67 | 1152 | 213 | 234 | 2623 | 69 | 9 |
| 67 | 737 | 125 | 118 | 2223 | 69 | 10 |
| 1756 | 10316 | 5832 | 1715 | 11827 | 419 | 418 |
| 2208 | 13978 | 4420 | 2839 | 17968 | 667 | 541 |
| 377141 | 212970 | 270886 | 187659 | 120398 | 16335 | 6397 |

Table 2: Results for the computation of Reach$_{\mathcal{A}}$ and G$_{\mathcal{A}}$. The Computation (resp. Storage) column gives the memory required to compute G$_{\mathcal{A}}$ (resp. to store G$_{\mathcal{A}}$).



(a) Time usage (ms) for *VIM plugins*.

(b) Mem. usage (kB) for *VIM plugins*.

(c) Time usage for *Azure*.

(d) Mem. usage for *Azure*.

(e) Time usage for *codecov*.

(f) Mem. usage for *codecov*.

Fig. 3: Results of validation benchmarks.

On all considered documents, both algorithms return the same classification output, even for the partially learned 1-SEVPA.

For our algorithm, we only measure the memory required to execute the algorithm, as we do not need to store the whole document to be able to process it. We also do not count the memory to store the 1-SEVPA and its key graph. As the classical algorithm must have the complete document stored in memory, we sum the RAM consumption for the document and for the algorithm itself. This is coherent to what happens in actual web-service handling: Whenever a new validation request is received, we would spawn a new subprocess that handles a specific document. Since the 1-SEVPA and its key graph are the same for all subprocesses, they would be loaded in a memory space shared by all processes.

Experimental results indicate that our algorithm exhibits good performance. Results for the three smaller schemas are not presented here to save space, while they are given in Figure 3 for *VIM plugins*, *Azure Functions Proxies*, and *codecov*. The blue (resp. red) crosses (resp. circles) give the values for our (resp. the classical) algorithm. The x-axis gives the size of each (abstracted) document.

For both *VIM plugins* and *Azure Functions Proxies*, our algorithm consumes less memory than the classical one. For these benchmarks, memory and time usage seemingly trade off as we see that our algorithm usually requires more time to validate a document — a majority of that time is spent computing the set $\text{Valid}(K, \mathit{Bad})$. This tradeoff, however, does not hold in general: There are schemas for which our algorithm performs better than the classical one, both in terms of time and memory, as it does not have to backtrack to validate a document, which reduces the time and memory space required.

For the *codecov* schema, we recall that the learning process was not completed, leading to an approximated 1-SEVPA with repeated keys in its key graph. This means that the computation of $\text{Valid}(K, \mathit{Bad})$ explores some invalid paths, increasing the memory and time consumed by our algorithm. Thus, it appears that, while a not completely learned 1-SEVPA can still be used in our algorithm, stopping the learning process early may increase the time and space required.

## 6 Future Work

As future work, one could focus on constructing the VPA directly from the schema, without going through a learning algorithm. While this task is easy if the schema does not contain Boolean operations, it is not yet clear how to proceed in the general case. Second, it could be worthwhile to compare our algorithm against an implementation of a classical algorithm used in the industry. This would require either to modify the industrial implementations to support abstractions, or to modify our algorithm to work on unabstracted JSON schemas. Third, in our validation approach, we decided to use a VPA accepting the JSON documents satisfying a fixed key order — thus requiring to use the key graph and its costly computation of the set $\text{Valid}(K, \mathit{Bad})$. It could be interesting to make additional experiments to compare this approach with one where we instead use a VPA accepting the JSON documents and all their key permutations — in this case, reasoning on the key graph would no longer be needed. Finally, motivated by obtaining efficient querying algorithms on XML trees, the authors of [26] have introduced the concept of mixed automata in a way to accept subsets of unranked trees where some nodes have ordered sons and some other have unordered sons. It would be interesting to adapt our validation algorithm to different formalisms of documents, such as the one of mixed automata.

**Data-Availability Statement.** The source code and experimental results that support the findings of this study are available in Zenodo with the identifier https://doi.org/10.5281/zenodo.7309690 [31].

## References

1. Richard A. DeMillo, Richard J. Lipton, Frederick G. Sayward: Hints on Test Data Selection: Help for the Practicing Programmer. Computer **11**(4), 34–41 (1978). https://doi.org/10.1109/C-M.1978.218136
2. Rajeev Alur, Viraj Kumar, Madhusudan, P., Mahesh Viswanathan: Congruences for Visibly Pushdown Languages. In: Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, Moti Yung (eds.) Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3580, pp. 1102–1114. Springer (2005). https://doi.org/10.1007/11523468_89
3. Rajeev Alur, Madhusudan, P.: Visibly pushdown languages. In: László Babai (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 202–211. ACM (2004). https://doi.org/10.1145/1007352.1007390
4. Dana Angluin: Learning Regular Sets from Queries and Counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
5. Iovka Boneva, Radu Ciucanu, Slawek Staworko: Schemas for Unordered XML on a DIME. Theory Comput. Syst. **57**(2), 337–376 (2015). https://doi.org/10.1007/s00224-014-9593-1
6. Tim Bray: The JavaScript Object Notation (JSON) Data Interchange Format. RFC **8259**, 1–16 (2017). https://doi.org/10.17487/RFC8259
7. Véronique Bruyère, Guillermo A. Pérez, Gaëtan Staquet: Learning Realtime One-Counter Automata. In: Dana Fisman, Grigore Rosu (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13243, pp. 244–262. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_13
8. John E. Hopcroft, Jeffrey D. Ullman: Introduction to Automata Theory, Languages and Computation, Second Edition. Addison-Wesley (2000)
9. Malte Isberner: Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis, Technical University Dortmund, Germany (2015), http://hdl.handle.net/2003/34282
10. Malte Isberner, Falk Howar, Bernhard Steffen: The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In: Borzoo Bonakdarpour, Scott A. Smolka (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26

11. Isberner, M., Howar, F., Steffen, B.: The Open-Source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 487–495. Springer International Publishing, Cham (2015)
12. Yue Jia, Mark Harman: An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Software Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
13. JSON Schema: JSON Schema website. Online (2015), https://json-schema.org
14. JSON.org: JSON.org website. Online (2013), https://www.json.org
15. Viraj Kumar, Madhusudan, P., Mahesh Viswanathan: Visibly pushdown automata for streaming XML. In: Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, Prashant J. Shenoy (eds.) Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007. pp. 1053–1062. ACM (2007). https://doi.org/10.1145/1242572.1242714
16. Wim Martens, Frank Neven, Matthias Niewerth, Thomas Schwentick: BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. ACM Trans. Database Syst. **42**(3), 15:1–15:42 (2017). https://doi.org/10.1145/3105960
17. Maik Merten, Bernhard Steffen, Falk Howar, Tiziana Margaria: Next Generation LearnLib. In: Parosh Aziz Abdulla, Rustan M. Leino, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6605, pp. 220–223. Springer (2011). https://doi.org/10.1007/978-3-642-19835-9_18
18. Matthias Niewerth, Thomas Schwentick: Reasoning About XML Constraints Based on XML-to-Relational Mappings. Theory Comput. Syst. **62**(8), 1826–1879 (2018). https://doi.org/10.1007/s00224-018-9846-5
19. Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, Domagoj Vrgoc: Foundations of JSON Schema. In: Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, Ben Y. Zhao (eds.) Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016. pp. 263–273. ACM (2016). https://doi.org/10.1145/2872427.2883029
20. Schema for Azure Functions Proxies. Online, https://json.schemastore.org/proxies.json
21. Schema for codecov configuration. Online, https://json.schemastore.org/codecov.json
22. Schema for VIM plugins. Online, https://json.schemastore.org/vim-addon-info.json
23. Schema for Visual Studio Code snippets. Online, https://raw.githubusercontent.com/Yash-Singh1/vscode-snippets-json-schema/main/schema.json
24. Thomas Schwentick: Foundations of XML Based on Logic and Automata: A Snapshot. In: Thomas Lukasiewicz, Attila Sali (eds.) Foundations of Information and Knowledge Systems - 7th International Symposium, FoIKS 2012, Kiel, Germany, March 5-9, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7153, pp. 23–33. Springer (2012). https://doi.org/10.1007/978-3-642-28472-4_2
25. Luc Segoufin, Victor Vianu: Validating Streaming XML Documents. In: Lucian Popa, Serge Abiteboul, Phokion G. Kolaitis (eds.) Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA. pp. 53–64. ACM (2002). https://doi.org/10.1145/543613.543622

26. Helmut Seidl, Thomas Schwentick, Anca Muscholl: Counting in trees. In: Jörg Flum, Erich Grädel, Thomas Wilke (eds.) Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]. Texts in Logic and Games, vol. 2, pp. 575–612. Amsterdam University Press (2008)
27. Staquet, G.: AutomataLib, https://github.com/DocSkellington/automatalib
28. Staquet, G.: JSON Schema Tools, https://github.com/DocSkellington/JSONSchemaTools
29. Staquet, G.: LearnLib, https://github.com/DocSkellington/learnlib
30. Staquet, G.: Validating JSON Documents With Learned VPA, https://github.com/DocSkellington/ValidatingJSONDocumentsWithLearnedVPA
31. Staquet, G.: Validating Streaming JSON Documents with Learned VPAs (Nov 2022). https://doi.org/10.5281/zenodo.7309689
32. Nguyen Van Tang: A Tighter Bound for the Determinization of Visibly Pushdown Automata. In: Axel Legay (ed.) Proceedings International Workshop on Verification of Infinite-State Systems, INFINITY 2009, Bologna, Italy, 31th August 2009. EPTCS, vol. 10, pp. 62–76 (2009). https://doi.org/10.4204/EPTCS.10.5