Verification of computer systems thanks to state machines

Gaëtan Staquet

Theoretical computer science
Computer Science Department
Science Faculty
University of Mons

Formal Techniques in Software Engineering
Computer Science Department
Science Faculty
University of Antwerp

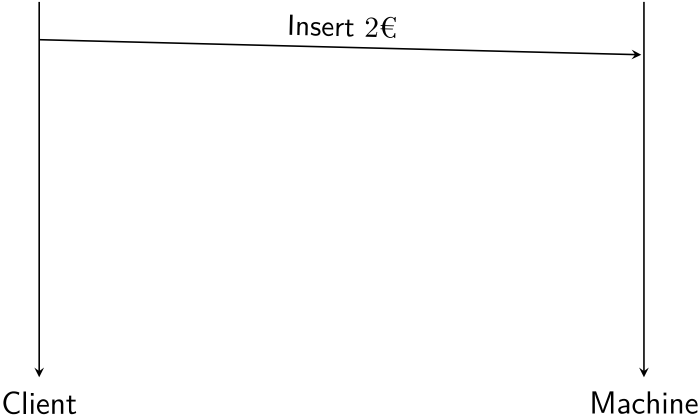
November 15, 2022

UMONS Université de Mons

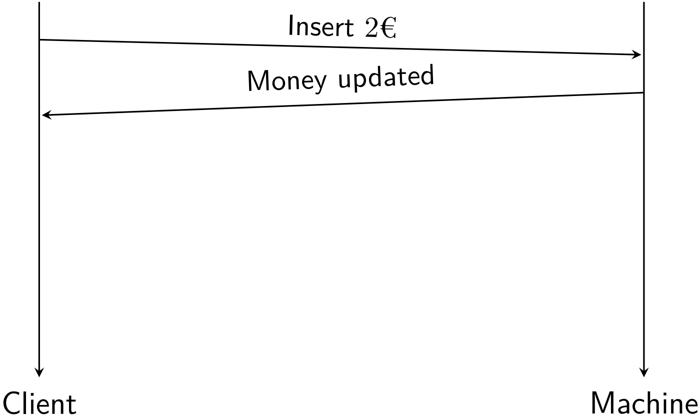
fnrs LA LIBERTÉ DE CHERCHER

University of Antwerp

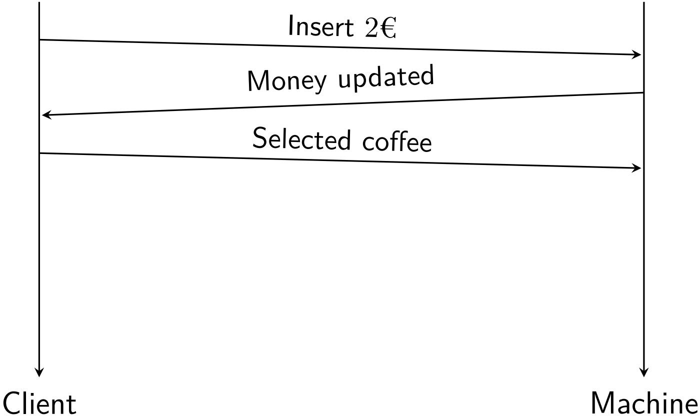
Coffee Machine – Correct execution



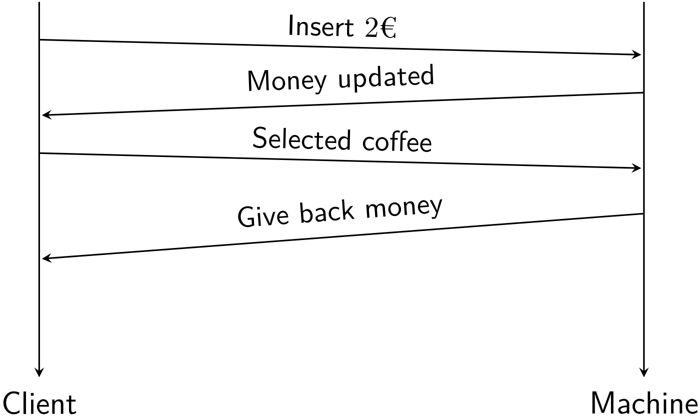
Coffee Machine – Correct execution



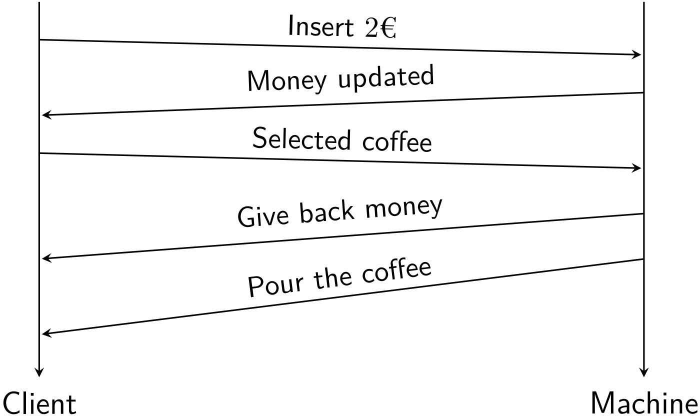
Coffee Machine – Correct execution



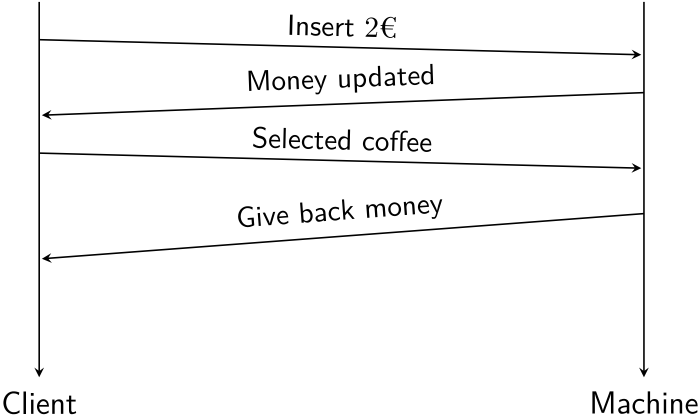
Coffee Machine – Correct execution



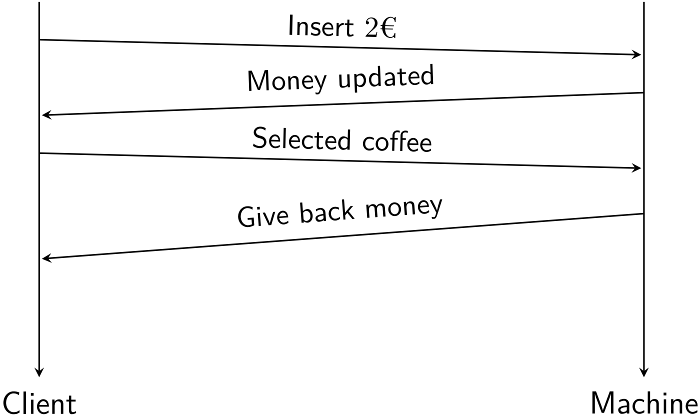
Coffee Machine – Correct execution



Coffee Machine – Error



Coffee Machine – Error



How can we detect the fault as soon as possible?

Detecting faults

Unit tests?

Detecting faults

Unit tests?

- Needs to implement the tests "manually".

Detecting faults

Unit tests?

- Needs to implement the tests "manually".
- Risk of forgetting important cases.

Detecting faults

Unit tests?

- Needs to implement the tests “manually”.
- Risk of forgetting important cases.
- Impossible to test everything.

Detecting faults

Unit tests?

- Needs to implement the tests "manually".
- Risk of forgetting important cases.
- Impossible to test everything.

$\hookrightarrow$ Does not prove the system is correct.

Detecting faults

Unit tests?

- Needs to implement the tests “manually”.
- Risk of forgetting important cases.
- Impossible to test everything.

$\hookrightarrow$ Does not prove the system is correct.

We will rely on formal methods.

Detecting faults

Unit tests?

- Needs to implement the tests "manually".
- Risk of forgetting important cases.
- Impossible to test everything.

$\hookrightarrow$ Does not prove the system is correct.

We will rely on formal methods.

Idea:

- Construct a model $\mathcal{M}$ of the system.
- Verify if $\mathcal{M}$ satisfies the desired properties, over all possible executions.

Detecting faults

Unit tests?

- Needs to implement the tests "manually".
- Risk of forgetting important cases.
- Impossible to test everything.

$\hookrightarrow$ Does not prove the system is correct.

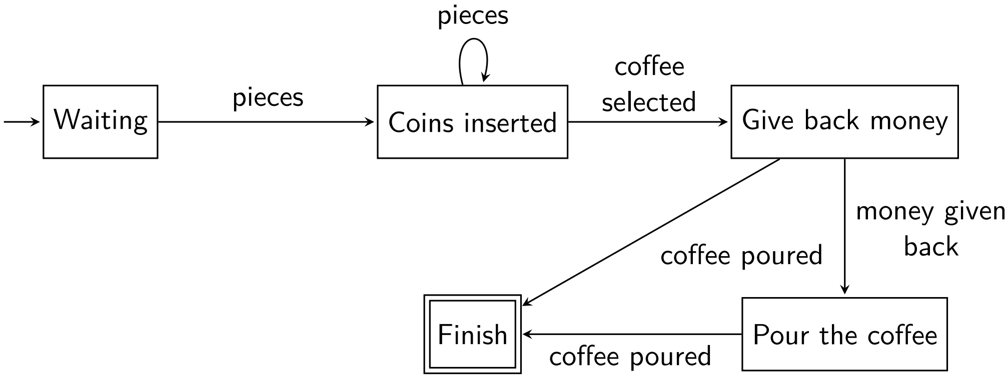
We will rely on formal methods.

Idea:

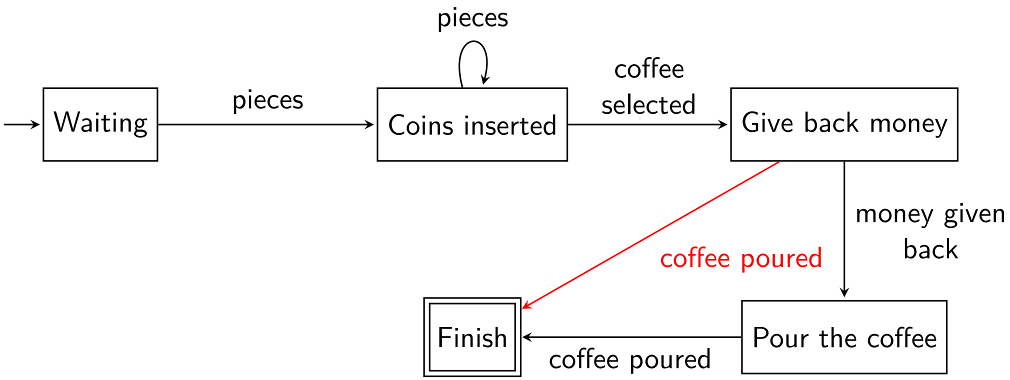
- Construct a model $\mathcal{M}$ of the system.
- Verify if $\mathcal{M}$ satisfies the desired properties, over all possible executions.

Here, we focus on the construction of the model.

A model for the coffee machine



A model for the coffee machine



Which model?

An alphabet, noted Σ, is a finite and non-empty set of symbols.

Example 1

$\Sigma = \{a, b\}$ is an alphabet.

Which model?

An alphabet, noted Σ, is a finite and non-empty set of symbols.
A word $w = a_1 a_2 \dots a_n$ $(n \in \mathbb{N})$ over an alphabet Σ is a finite sequence of symbols, $a_i \in \Sigma$. The empty word is denoted by ε.

Example 1

$\Sigma = \{a, b\}$ is an alphabet.
$w = ababb$ is a word over Σ.

Which model?

An alphabet, noted Σ, is a finite and non-empty set of symbols.
A word $w = a_1 a_2 \dots a_n$ $(n \in \mathbb{N})$ over an alphabet Σ is a finite sequence of symbols, $a_i \in \Sigma$. The empty word is denoted by ε.
A language L over an alphabet Σ is a set of words.

Example 1

$\Sigma = \{a, b\}$ is an alphabet.
$w = ababb$ is a word over Σ.
$L' = \{\varepsilon, a, b\}$ and $L = \{w \mid w$ has an even number of a and an odd number of $b\}$ are two languages over Σ.

Which model?

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Σ an alphabet;

Which model?

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Σ an alphabet;
- Q a finite set of states;

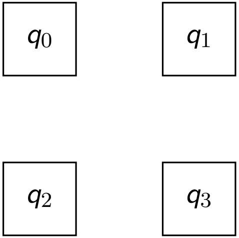


Figure 1: A DFA $\mathcal{A}$.

Which model?

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Σ an alphabet;
- Q a finite set of states;
- $\delta : (Q \times \Sigma) \to Q$ a transition function;

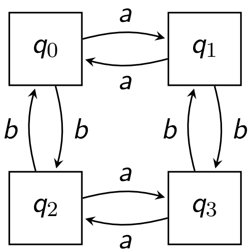


Figure 1: A DFA $\mathcal{A}$.

Which model?

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Σ an alphabet;
- Q a finite set of states;
- $\delta : (Q \times \Sigma) \to Q$ a transition function;
- $q_0 \in Q$ the initial state;

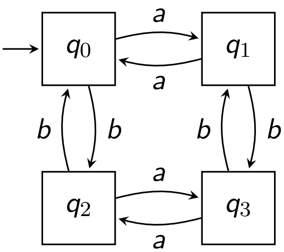


Figure 1: A DFA $\mathcal{A}$.

Which model?

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Σ an alphabet;
- Q a finite set of states;
- $\delta : (Q \times \Sigma) \to Q$ a transition function;
- $q_0 \in Q$ the initial state;
- $F \subseteq Q$ the set of final states.

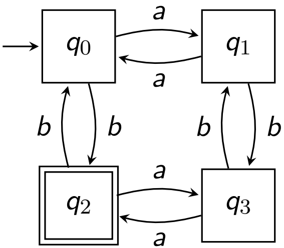


Figure 1: A DFA $\mathcal{A}$.

Which model?

Let $w = a_1 a_2 \ldots, a_n \in \Sigma^*$. The run of $\mathcal{A}$ over w is the sequence of states

$$p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} p_3 \xrightarrow{a_3} \ldots \xrightarrow{a_n} p_{n+1}$$

such that $p_1 = q_0$ and $\forall i, \delta(p_i, a_i) = p_{i+1}$.

Example 2

Let $w = ababb$. The corresponding run is

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_3 \xrightarrow{a} q_2 \xrightarrow{b} q_0 \xrightarrow{b} q_2.$$

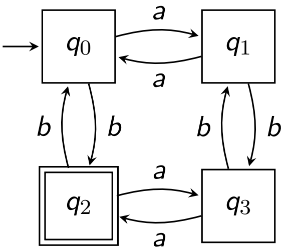


Figure 1: A DFA $\mathcal{A}$.

Which model?

Let $w = a_1 a_2 \ldots, a_n \in \Sigma^*$. The run of $\mathcal{A}$ over w is the sequence of states

$$p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} p_3 \xrightarrow{a_3} \ldots \xrightarrow{a_n} p_{n+1}$$

such that $p_1 = q_0$ and $\forall i, \delta(p_i, a_i) = p_{i+1}$.
If $p_{n+1} \in F$, then w is accepted by $\mathcal{A}$.

Example 2

Let $w = ababb$. The corresponding run is

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_3 \xrightarrow{a} q_2 \xrightarrow{b} q_0 \xrightarrow{b} q_2,$$

and w is accepted by $\mathcal{A}$.

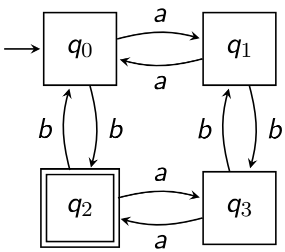


Figure 1: A DFA $\mathcal{A}$.

Which model?

The language of $\mathcal{A}$ is the set of all accepted words, i.e.,

$$\mathcal{L}(\mathcal{A}) = \{w \mid \exists p \in F, q_0 \xrightarrow{w} p\}.$$

Example 3

The language of $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{w \mid w \text{ has an even number of } a \text{ and an odd number of } b\}.$$

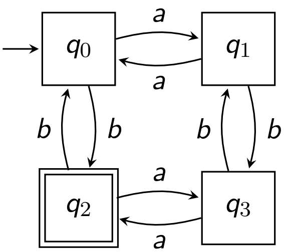


Figure 1: A DFA $\mathcal{A}$.

Infinite table

Let $L = \{w \mid w \text{ has an even number of } a \text{ and an odd number of } b\}$.

Let $u \in \Sigma^*$. For all $w \in \Sigma^*$, we check whether $uw \in L$.
We construct a table where the rows are the u and the columns the w.

Infinite table

Let $L = \{w \mid w \text{ has an even number of } a \text{ and an odd number of } b\}$.

	ε	a	b	aa	ab	ba	bb	$\dots$
ε	0	0	1	0	0	0	0	$\dots$
a	0	0	0	0	1	1	0	$\dots$
b	1	0	0	1	0	0	1	$\dots$
aa	0	0	1	0	0	0	0	$\dots$
ab	0	1	0	0	0	0	0	$\dots$
ba	0	1	0	0	0	0	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Infinite table

Let $L = \{w \mid w \text{ has an even number of } a \text{ and an odd number of } b\}$.

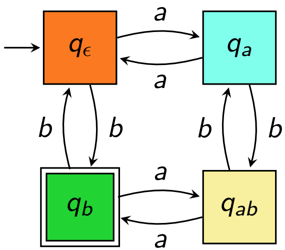
	ε	a	b	aa	ab	ba	bb	$\dots$
ε	0	0	1	0	0	0	0	$\dots$
a	0	0	0	0	1	1	0	$\dots$
b	1	0	0	1	0	0	1	$\dots$
aa	0	0	1	0	0	0	0	$\dots$
ab	0	1	0	0	0	0	0	$\dots$
ba	0	1	0	0	0	0	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The table contains in fact four different rows.

Infinite table

Let $L = \{w \mid w$ has an even number of a and an odd number of $b\}$.

	ε	a	b	aa	ab	ba	bb	...
ε	0	0	1	0	0	0	0	...
a	0	0	0	0	1	1	0	...
b	1	0	0	1	0	0	1	...
aa	0	0	1	0	0	0	0	...
ab	0	1	0	0	0	0	0	...
ba	0	1	0	0	0	0	0	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

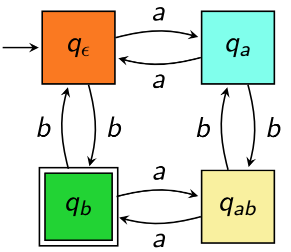


The table contains in fact four different rows.

Infinite table

Let $L = \{w \mid w \text{ has an even number of } a \text{ and an odd number of } b\}$.

	ε	a	b	aa	ab	ba	bb	$\dots$
ε	0	0	1	0	0	0	0	$\dots$
a	0	0	0	0	1	1	0	$\dots$
b	1	0	0	1	0	0	1	$\dots$
aa	0	0	1	0	0	0	0	$\dots$
ab	0	1	0	0	0	0	0	$\dots$
ba	0	1	0	0	0	0	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$



The table contains in fact four different rows.
$\hookrightarrow$ A finite table is enough.

How to learn a table?

Figure 2: Angluin's framework.[1]

[1] Angluin, "Learning Regular Sets from Queries and Counterexamples", 1987.

How to learn a table?

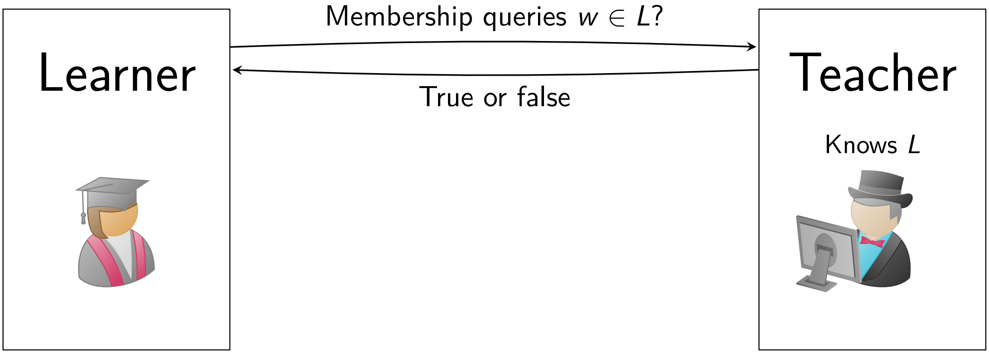


Figure 2: Angluin's framework.[1]

[1] Angluin, "Learning Regular Sets from Queries and Counterexamples", 1987.

How to learn a table?

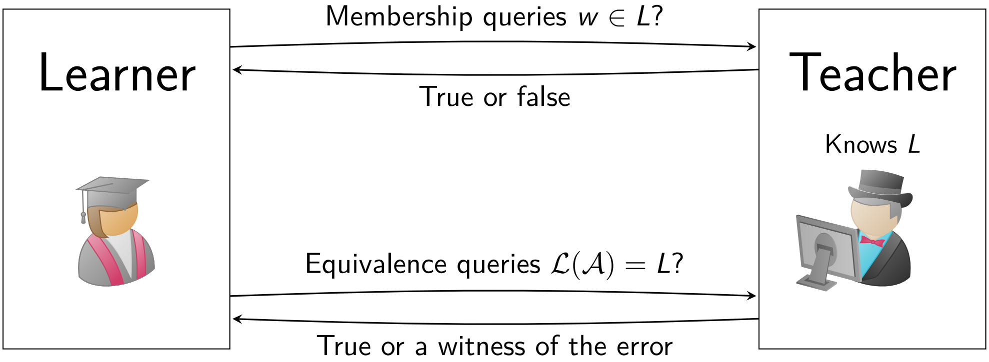


Figure 2: Angluin's framework.[1]

[1] Angluin, "Learning Regular Sets from Queries and Counterexamples", 1987.

Back to the coffe

How does the teacher work, in practical cases?

- Membership queries: execute the system on w and provide the answer.

Back to the coffe

How does the teacher work, in practical cases?

- Membership queries: execute the system on w and provide the answer.
- Equivalence queries:
  - If we can manipulate the system as a black box, then we can approximate the equivalence queries.

Back to the coffe

How does the teacher work, in practical cases?

- Membership queries: execute the system on w and provide the answer.
- Equivalence queries:
  - If we can manipulate the system as a black box, then we can approximate the equivalence queries.
  - If we know how the system behaves (white box), then the equivalence queries can be more precise.

Back to the coffe

How does the teacher work, in practical cases?

- Membership queries: execute the system on w and provide the answer.
- Equivalence queries:
  - If we can manipulate the system as a black box, then we can approximate the equivalence queries.
  - If we know how the system behaves (white box), then the equivalence queries can be more precise.
  - We can mix both approaches (grey box).

Back to the coffe

How does the teacher work, in practical cases?

- Membership queries: execute the system on w and provide the answer.
- Equivalence queries:
  - If we can manipulate the system as a black box, then we can approximate the equivalence queries.
  - If we know how the system behaves (white box), then the equivalence queries can be more precise.
  - We can mix both approaches (grey box).

$\hookrightarrow$ It depends on the exact problem.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

We want to verify that the document satisfies some constraints.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

"title" $\mapsto$ string of characters

"place" $\mapsto$ object such that

- "town" $\mapsto$ string of characters
- "country" $\mapsto$ string of characters

"date" $\mapsto$ array of integers

We want to verify that the document satisfies some constraints.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

"title" $\mapsto$ string of characters
"place" $\mapsto$ object such that
    "town" $\mapsto$ string of characters
    "country" $\mapsto$ string of characters
"date" $\mapsto$ array of integers

We want to verify that the document satisfies some constraints.

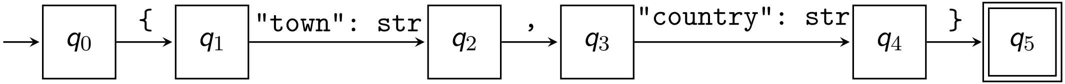


Figure 3: An automaton for the value of "place".

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

"title" $\mapsto$ string of characters

"place" $\mapsto$ object such that

"town" $\mapsto$ string of characters

"country" $\mapsto$ string of characters

"date" $\mapsto$ array of integers

- An object is a non-ordered collection of key-value paires.

[a] Bruyère, Pérez, and Staquet, *Validating JSON Documents with Learned VPAs*, 2022.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

"title" $\mapsto$ string of characters
"place" $\mapsto$ object such that
  "town" $\mapsto$ string of characters
  "country" $\mapsto$ string of characters
"date" $\mapsto$ array of integers

- An object is a non-ordered collection of key-value paires.
- An array is an ordered collection of values.

[a] Bruyère, Pérez, and Staquet, *Validating JSON Documents with Learned VPAs*, 2022.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

`"title"` $\mapsto$ string of characters

`"place"` $\mapsto$ object such that

  `"town"` $\mapsto$ string of characters

  `"country"` $\mapsto$ string of characters

`"date"` $\mapsto$ array of integers

- An object is a non-ordered collection of key-value paires.
- An array is an ordered collection of values.

Our approach[a]:

[a] Bruyère, Pérez, and Staquet, *Validating JSON Documents with Learned VPAs*, 2022.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

`"title"` $\mapsto$ string of characters
`"place"` $\mapsto$ object such that
`"town"` $\mapsto$ string of characters
`"country"` $\mapsto$ string of characters
`"date"` $\mapsto$ array of integers

- An object is a non-ordered collection of key-value paires.
- An array is an ordered collection of values.

Our approach[a]:

- We learn an automaton $\mathcal{A}$ with a fixed order on the keys.

[a] Bruyère, Pérez, and Staquet, *Validating JSON Documents with Learned VPAs*, 2022.

JSON Documents

```
{
  "title": "Verification by state machines",
  "place": {
    "town": "Mons",
    "country": "Belgium"
  },
  "date": [15, 11, 2022]
}
```

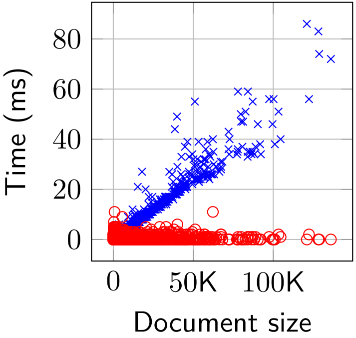
"title" $\mapsto$ string of characters
"place" $\mapsto$ object such that
    "town" $\mapsto$ string of characters
    "country" $\mapsto$ string of characters
"date" $\mapsto$ array of integers

- An object is a non-ordered collection of key-value paires.
- An array is an ordered collection of values.

Our approach[a]:

- We learn an automaton $\mathcal{A}$ with a fixed order on the keys.
- We abstract $\mathcal{A}$ to allow any order.

[a] Bruyère, Pérez, and Staquet, *Validating JSON Documents with Learned VPAs*, 2022.



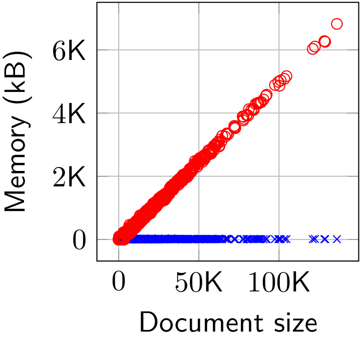


Figure 4: Experimental results for our JSON documents validation algorithm. Blue crosses given the values for our algorithm, and the red circles for the “classical” algorithm.

References I

Angluin, Dana. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6. URL: https://doi.org/10.1016/0890-5401(87)90052-6.

Bruyère, V., G. A. Pérez, and G. Staquet. *Validating JSON Documents with Learned VPAs*. Pre-print. Soumis à TACAS 2023. F.R.S.-FNRS, Universités de Mons et d'Anvers, 2022.