Active Learning of Automata with Resources

Public PhD Defense

Gaëtan Staquet

Theoretical Computer Science
University of Mons

Formal Techniques in Software Engineering
University of Antwerp

September 11, 2024

Part I – Preliminaries

Base definitions

Sometimes, an old computer system has to be re-implemented in a modern language.
For instance, see the TRACTOR (TRanslating All C TO Rust) of DARPA.[1]

[1] `https://www.darpa.mil/program/translating-all-c-to-rust`

Sometimes, an old computer system has to be re-implemented in a modern language. For instance, see the TRACTOR (TRanslating All C TO Rust) of DARPA.[1]

Two approaches:

Do the conversion by hand and then check that both implementations are **equivalent**, *i.e.*, they have the same behavior.

Do the conversion automatically using tools that ensure the old and new systems are **equivalent**.

[1] `https://www.darpa.mil/program/translating-all-c-to-rust`

Sometimes, an old computer system has to be re-implemented in a modern language. For instance, see the TRACTOR (TRanslating All C TO Rust) of DARPA.[1]

Two approaches:

Do the conversion by hand and then check that both implementations are **equivalent**, *i.e.*, they have the same behavior.

Do the conversion automatically using tools that ensure the old and new systems are **equivalent**.

In any case, use **model checking** tools:

1. Abstract the system into a **model**.
2. Verify that the model satisfies some **properties**.

[1] `https://www.darpa.mil/program/translating-all-c-to-rust`

Goals of this thesis:

- New tools for constructing two types of models:
  - one that can **count**, and
  - one that can enforce **timing** constraints.
- A whole model checking algorithm for a file format.

For our running example, we consider a (homemade) network protocol.

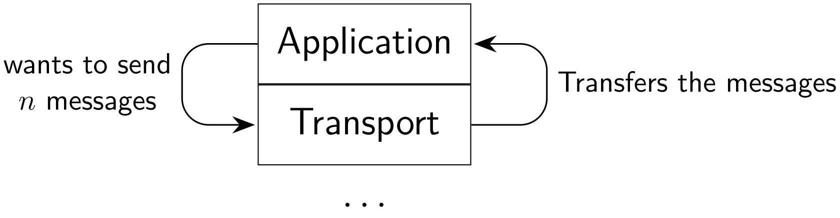


Figure 1: A part of the network stack.

The application can **freely** choose the value of n.

Figure 2: The sender sends n messages to the receiver.

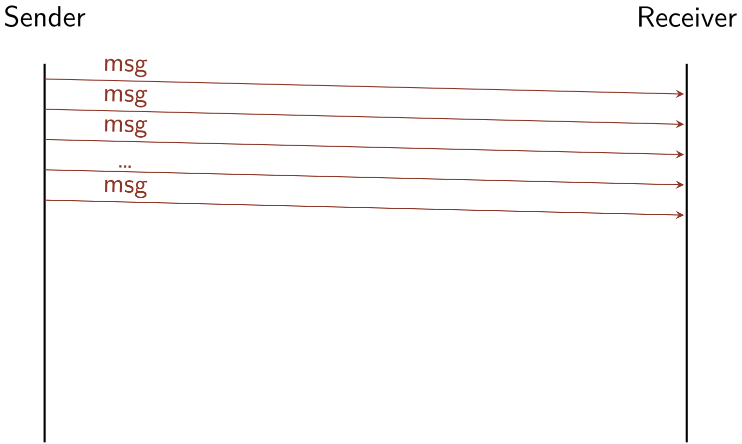


Figure 2: The sender sends n messages to the receiver.

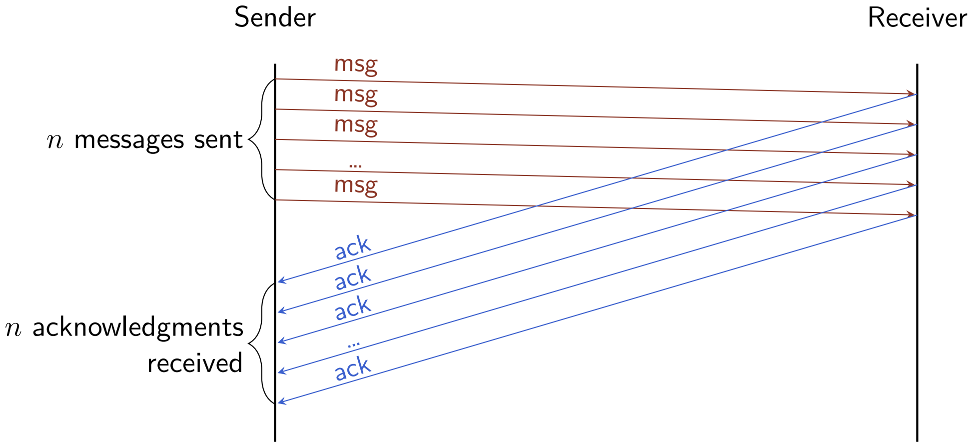


Figure 2: The sender sends n messages to the receiver.

Figure 3: The sender sends n messages to the receiver, who sometimes answers with messages.

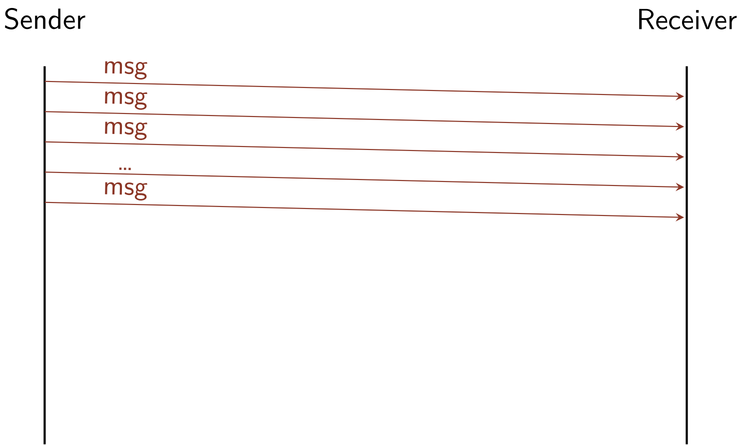


Figure 3: The sender sends n messages to the receiver, who sometimes answers with messages.

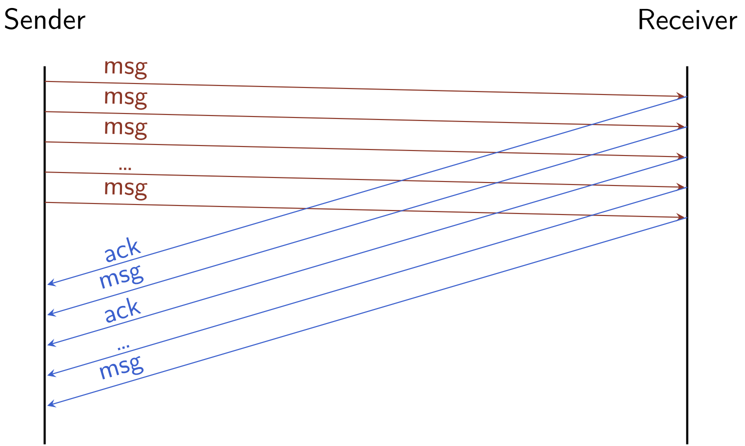


Figure 3: The sender sends n messages to the receiver, who sometimes answers with messages.

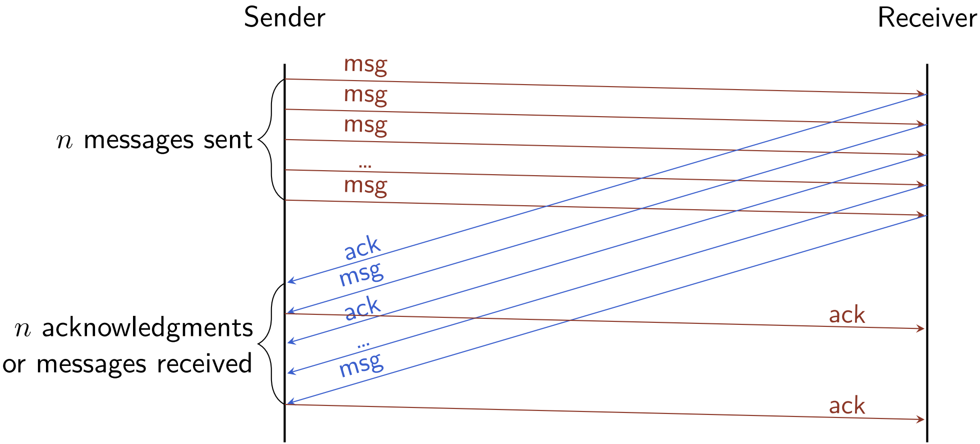


Figure 3: The sender sends n messages to the receiver, who sometimes answers with messages.

Figure 4: The sender sends n messages to the receiver but some of them are lost.

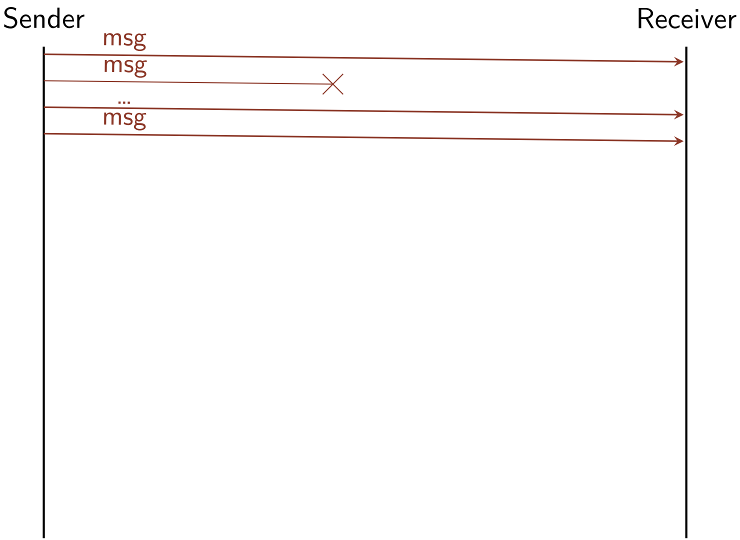


Figure 4: The sender sends n messages to the receiver but some of them are lost.

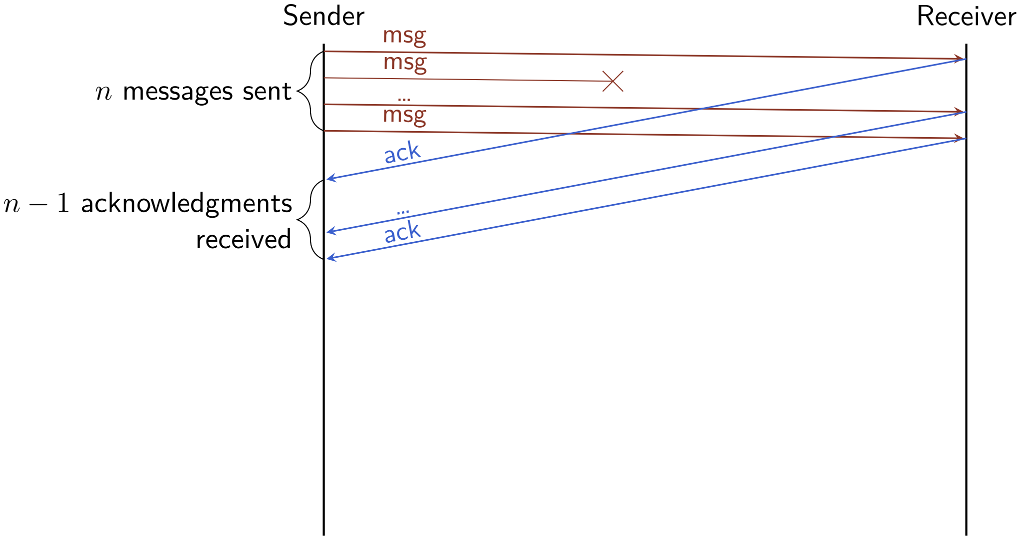


Figure 4: The sender sends n messages to the receiver but some of them are lost.

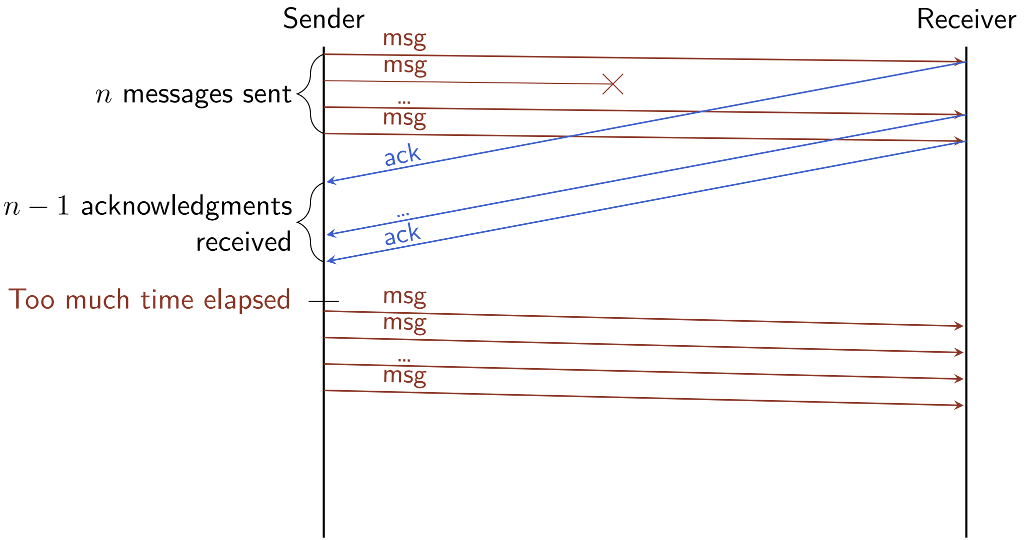


Figure 4: The sender sends n messages to the receiver but some of them are lost.

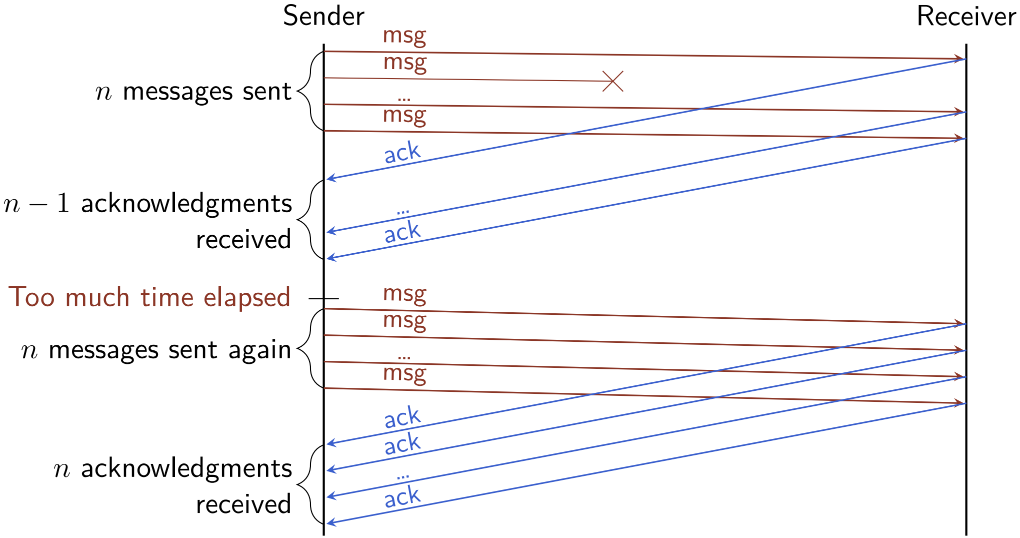


Figure 4: The sender sends n messages to the receiver but some of them are lost.

Part II – Learning models

Focus on how to infer models from a system

3. Automata

4. Counting

5. Timing constraints

We have a system we can interact with but we do not know how it works internally. We want to learn a **model** of the system.

Question. What kind of models? $\rightsquigarrow$ **Automata**.

We have a system we can interact with but we do not know how it works internally. We want to learn a **model** of the system.

Question. What kind of models? $\rightsquigarrow$ **Automata**.

A **word** is a sequence of symbols, *e.g.*, "msg msg done ack" is a word.

A **language** is a set of words.

We have a system we can interact with but we do not know how it works internally. We want to learn a **model** of the system.

> **Question.** What kind of models? $\rightsquigarrow$ **Automata**.

A **word** is a sequence of symbols, *e.g.*, "msg msg done ack" is a word.

A **language** is a set of words.

An **automaton** $\mathcal{A}$ is a model that computes whether a word is **accepted**.
The set of all words accepted by $\mathcal{A}$ is called the **language of $\mathcal{A}$**, denoted by $\mathcal{L}(\mathcal{A})$.

A **finite automaton** (**FA**, in short) is a “simple” automaton.

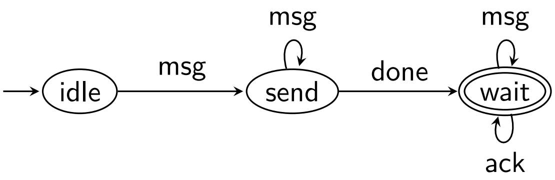


Figure 5: An FA for our network protocol.

A **finite automaton** (**FA**, in short) is a “simple” automaton.

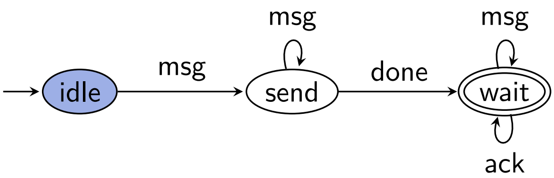


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle}$$

A **finite automaton** (**FA**, in short) is a “simple” automaton.

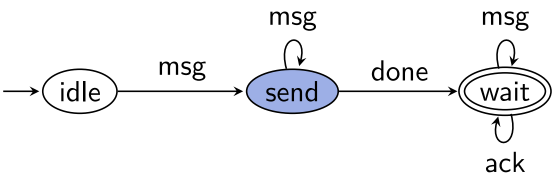


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle} \xrightarrow{\text{msg}} \text{send}$$

A **finite automaton** (**FA**, in short) is a "simple" automaton.

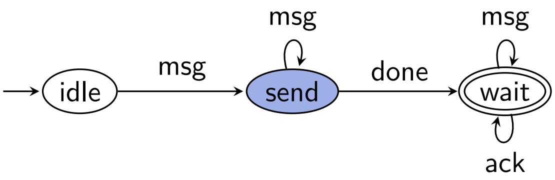


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{msg}} \text{send}$$

A **finite automaton** (**FA**, in short) is a “simple” automaton.

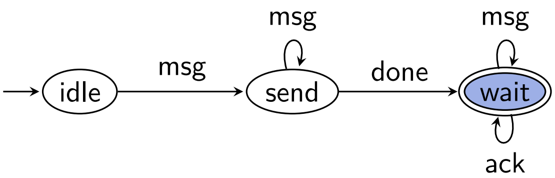


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{done}} \text{wait}$$

A **finite automaton** (**FA**, in short) is a "simple" automaton.

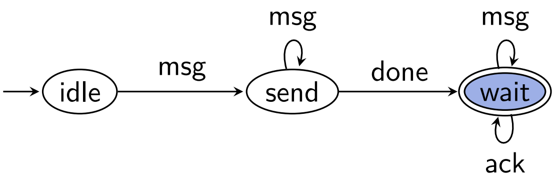


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{done}} \text{wait} \xrightarrow{\mathsf{ack}} \text{wait}.$$

A **finite automaton** (**FA**, in short) is a “simple” automaton.

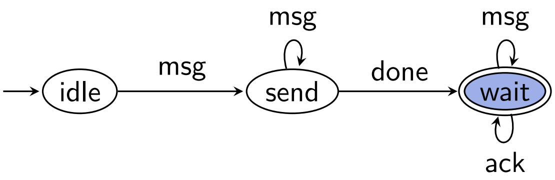


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\mathrm{idle} \xrightarrow{\mathsf{msg}} \mathrm{send} \xrightarrow{\mathsf{msg}} \mathrm{send} \xrightarrow{\mathsf{done}} \mathrm{wait} \xrightarrow{\mathsf{ack}} \mathrm{wait}.$$

So, “msg msg done ack” is **accepted**.

A **finite automaton** (**FA**, in short) is a "simple" automaton.

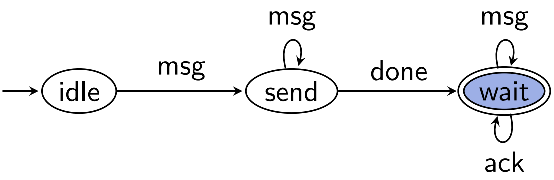


Figure 5: An FA for our network protocol.

A **run** of an FA is a sequence of states and symbols, *e.g.*,

$$\text{idle} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{msg}} \text{send} \xrightarrow{\mathsf{done}} \text{wait} \xrightarrow{\mathsf{ack}} \text{wait}.$$

So, "msg msg done ack" is **accepted**.

The language of an FA is called a **regular language**.

A **finite automaton** (**FA**, in short) is a “simple” automaton.

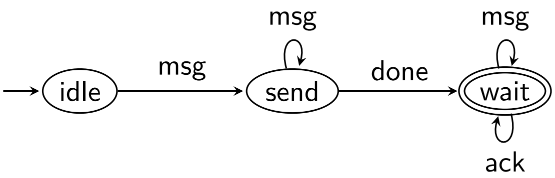


Figure 5: An FA for our network protocol.

This FA is not **expressive** enough for our network protocol:

- impossible to **count** towards an *a priori* unknown integer,
- impossible to measure **time**.

Hence, it is a **bold abstraction** of the system.

Question. How to infer an automaton from a system? $\rightsquigarrow$ **Active automata learning**.

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

Question. How to infer an automaton from a system? $\rightsquigarrow$ **Active automata learning**.

If the system can be abstracted into an FA, use the L^* learning algorithm.[2]

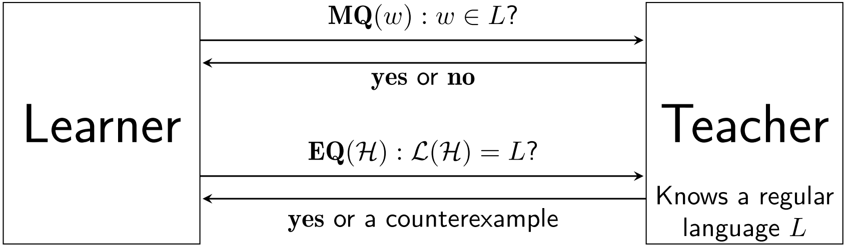


Figure 6: The Angluin's framework for learning FAs.

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

1. Ask membership queries over some words:

$$\begin{aligned} &\text{msg} && \rightsquigarrow && \textbf{no} \\ &\text{msg done} && \rightsquigarrow && \textbf{yes} \\ &\text{msg ack done} && \rightsquigarrow && \textbf{yes} \end{aligned}$$

1. Ask membership queries over some words:

msg	$\rightsquigarrow$	**no**
msg done	$\rightsquigarrow$	**yes**
msg ack done	$\rightsquigarrow$	**yes**

2. Build an FA and ask an equivalence query.

1. Ask membership queries over some words:

msg	$\rightsquigarrow$	**no**
msg done	$\rightsquigarrow$	**yes**
msg ack done	$\rightsquigarrow$	**yes**

2. Build an FA and ask an equivalence query.
3. If the answer is yes, stop. Otherwise, exploit the counterexample and repeat.

Theorem 1 (Angluin, "Learning Regular Sets from Queries and Counterexamples", 1987). *Let $\mathcal{A}$ be an FA accepting L and ℓ be the length of the longest counterexample. The L^* algorithm can learn an FA accepting L by asking a number of queries* ***polynomial*** *in the size of $\mathcal{A}$ and in ℓ.*

Theorem 1 (Angluin, "Learning Regular Sets from Queries and Counterexamples", 1987). *Let $\mathcal{A}$ be an FA accepting L and ℓ be the length of the longest counterexample. The L^* algorithm can learn an FA accepting L by asking a number of queries* ***polynomial*** *in the size of $\mathcal{A}$ and in ℓ.*

Again, an FA is not always expressive enough.
Structure of this part:

1. Augment automata with a **counter**.
2. Augment automata with a way to measure **time**.

3. Automata

4. Counting

5. Timing constraints

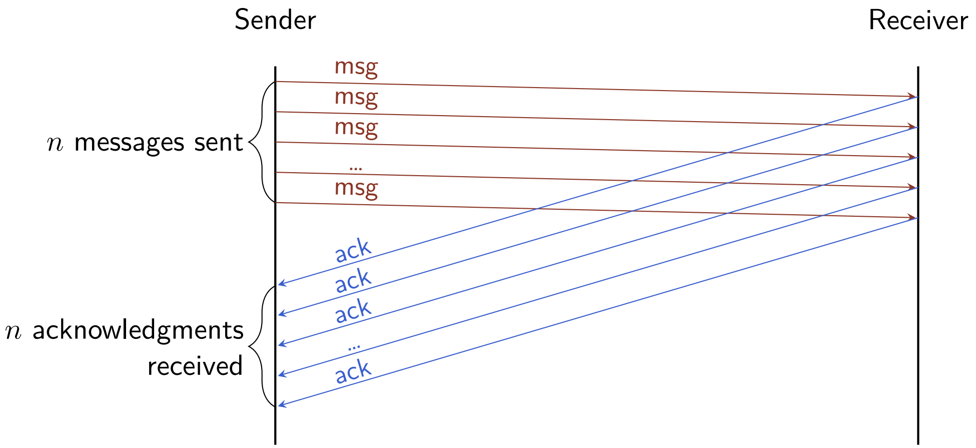


Figure 7: The network protocol must **count** the number of messages that is *a priori* unknown.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

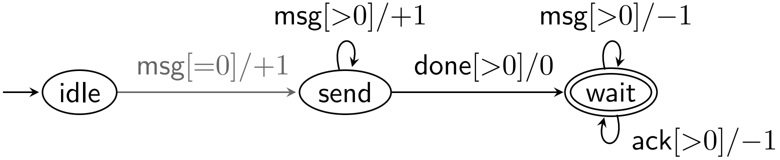


Figure 8: An ROCA for our network protocol.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

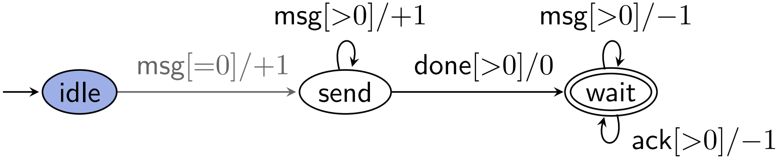


Figure 8: An ROCA for our network protocol.

$(\text{idle}, 0)$

So, “msg msg done ack ack” is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

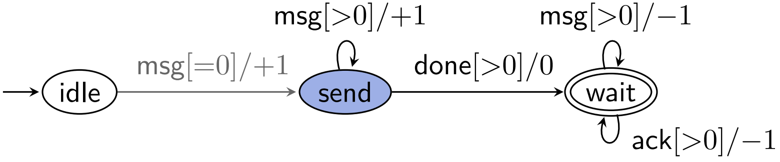


Figure 8: An ROCA for our network protocol.

$$(\text{idle}, 0) \xrightarrow{\text{msg}} (\text{send}, 1)$$

So, "msg msg done ack ack" is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

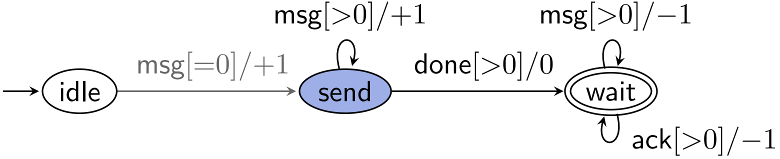


Figure 8: An ROCA for our network protocol.

$$(\text{idle}, 0) \xrightarrow{\text{msg}} (\text{send}, 1) \xrightarrow{\text{msg}} (\text{send}, 2)$$

So, "msg msg done ack ack" is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

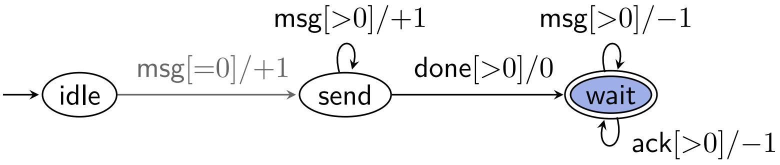


Figure 8: An ROCA for our network protocol.

$$(\text{idle}, 0) \xrightarrow{\text{msg}} (\text{send}, 1) \xrightarrow{\text{msg}} (\text{send}, 2)$$
$$\xrightarrow{\text{done}} (\text{wait}, 2)$$

So, “msg msg done ack ack” is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

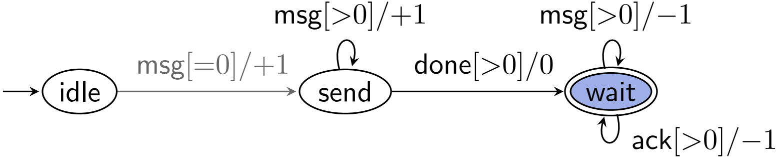


Figure 8: An ROCA for our network protocol.

$$(\mathrm{idle}, 0) \xrightarrow{\text{msg}} (\mathrm{send}, 1) \xrightarrow{\text{msg}} (\mathrm{send}, 2)$$
$$\xrightarrow{\text{done}} (\mathrm{wait}, 2) \xrightarrow{\text{ack}} (\mathrm{wait}, 1)$$

So, "msg msg done ack ack" is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

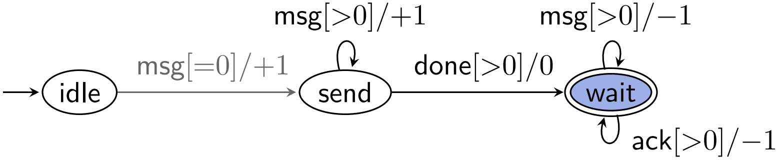


Figure 8: An ROCA for our network protocol.

$$(\mathrm{idle}, 0) \xrightarrow{\textsf{msg}} (\mathrm{send}, 1) \xrightarrow{\textsf{msg}} (\mathrm{send}, 2) \xrightarrow{\textsf{done}} (\mathrm{wait}, 2) \xrightarrow{\textsf{ack}} (\mathrm{wait}, 1) \xrightarrow{\textsf{ack}} (\mathrm{wait}, 0).$$

So, “msg msg done ack ack” is accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

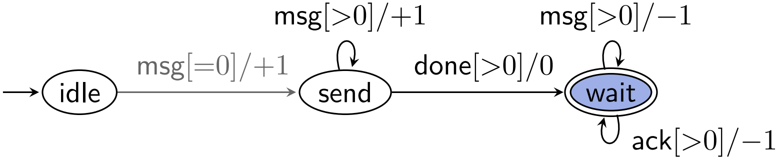


Figure 8: An ROCA for our network protocol.

$$(\mathrm{idle}, 0) \xrightarrow{\text{msg}} (\mathrm{send}, 1) \xrightarrow{\text{msg}} (\mathrm{send}, 2) \xrightarrow{\text{done}} (\mathrm{wait}, 2) \xrightarrow{\text{ack}} (\mathrm{wait}, 1) \xrightarrow{\text{ack}} (\mathrm{wait}, 0).$$

So, "msg msg done ack ack" is accepted.

$$(\mathrm{idle}, 0) \xrightarrow{\text{msg}} (\mathrm{send}, 1) \xrightarrow{\text{msg}} (\mathrm{send}, 2) \xrightarrow{\text{done}} (\mathrm{wait}, 2) \xrightarrow{\text{ack}} (\mathrm{wait}, 1).$$

So, "msg msg done ack" is **not** accepted.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

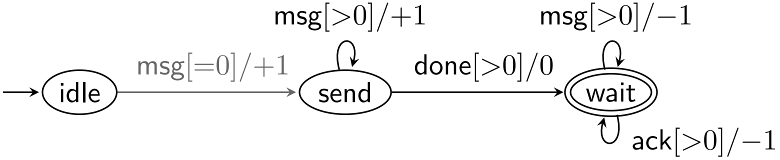


Figure 8: An ROCA for our network protocol.

We cannot measure **time**. This is again an **abstraction**.

We need to **count**. $\rightsquigarrow$ Realtime **one-counter** automata (**ROCA**, in short).

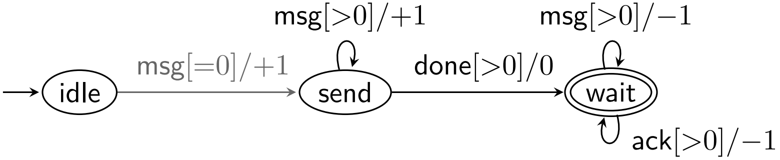


Figure 8: An ROCA for our network protocol.

We cannot measure **time**. This is again an **abstraction**.

Question. How to learn an ROCA from a system?

Hard task: deducing how the counter must change.

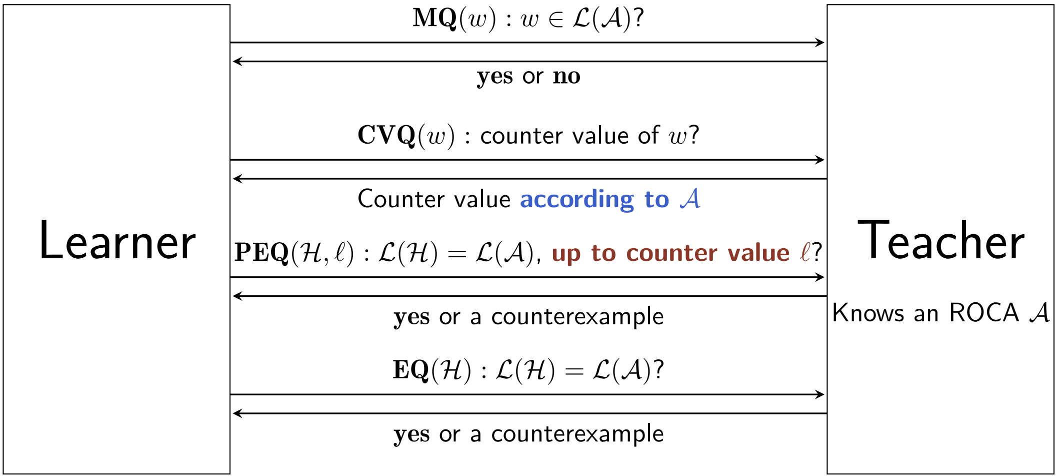


Figure 9: Adapted Angluin's framework for ROCAs.

Theorem 2. *Let $\mathcal{A}$ be the teacher's ROCA and ℓ the length of the longest counterexample. The L^*_{ROCA} algorithm can learn an ROCA equivalent to $\mathcal{A}$ and requires*

- *a number of (partial) equivalence queries* ***polynomial*** *in the size of $\mathcal{A}$ and in ℓ, and*
- *a number of membership and counter values* ***exponential*** *in the size of $\mathcal{A}$ and in ℓ.*

Theorem 2. *Let $\mathcal{A}$ be the teacher's ROCA and ℓ the length of the longest counterexample. The L^*_{ROCA} algorithm can learn an ROCA equivalent to $\mathcal{A}$ and requires*

- *a number of (partial) equivalence queries* ***polynomial*** *in the size of $\mathcal{A}$ and in ℓ, and*
- *a number of membership and counter values* ***exponential*** *in the size of $\mathcal{A}$ and in ℓ.*

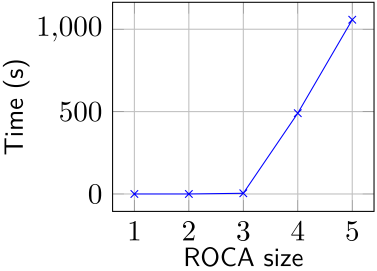
Main difficulty. Ensuring that the algorithm eventually stops when figuring out how the counter evolves.

We can only ask a counter value over w only when w satisfies some constraints. Making sure that we only consider finitely many w necessitates some work.

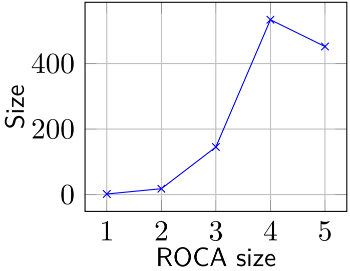
We implemented L^*_{ROCA} in Java using AutomataLib and LearnLib.[3]
Useful to identify and understand the difficulties.

We evaluated the performance on randomly generated ROCAs (and on some more concrete examples skipped here).

[3] Isberner, Howar, and Steffen, "The Open-Source LearnLib - A Framework for Active Automata Learning", 2015.



(a) Mean of the total time taken by L^*_{ROCA}.



(b) Mean of the final size of the data structure.

Figure 10: Results for the benchmarks based on random ROCAs.

3. Automata

4. Counting

5. Timing constraints

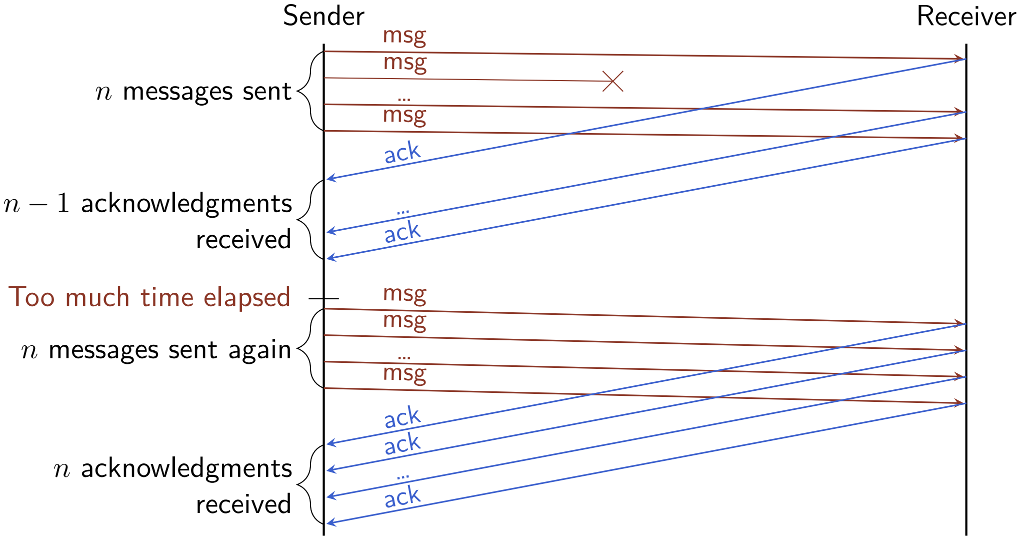


Figure 11: Our network protocol needs to measure **time**.

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

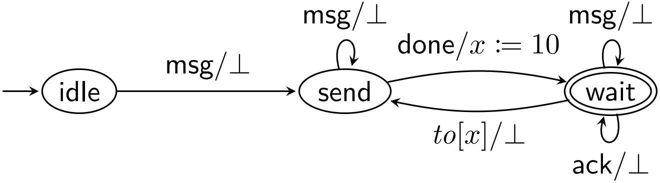


Figure 12: An AwT for an **abstraction** of our network protocol.

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

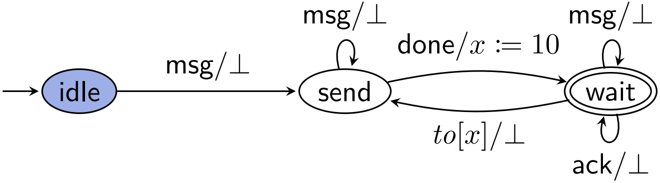


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

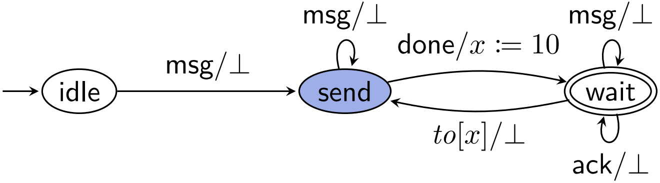


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\mathrm{idle}, \emptyset) \xrightarrow{\mathsf{msg}} (\mathrm{send}, \emptyset)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

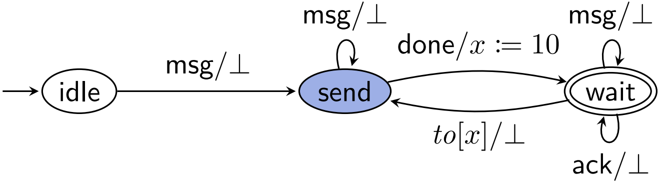


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset) \xrightarrow{\text{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset)$$

We must measure **time** and do something once a given amount of time has elapsed.
↪ Automata with **timers** (**AwT**, for short).

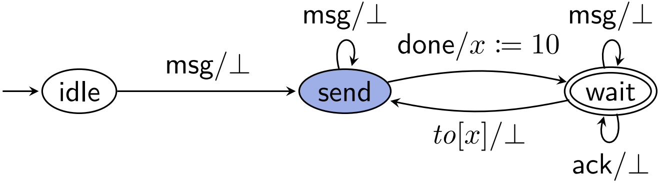


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset) \xrightarrow{\text{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\text{msg}} (\text{send}, \emptyset)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

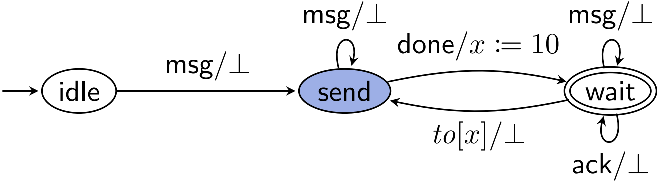


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset) \xrightarrow{\textsf{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\textsf{msg}} (\text{send}, \emptyset)$$
$$\xrightarrow{0.5} (\text{send}, \emptyset)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

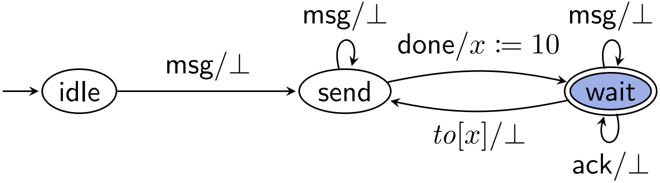


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset) \xrightarrow{\textsf{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\textsf{msg}} (\text{send}, \emptyset)$$
$$\xrightarrow{0.5} (\text{send}, \emptyset) \xrightarrow{\textsf{done}} (\text{wait}, x = 10)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

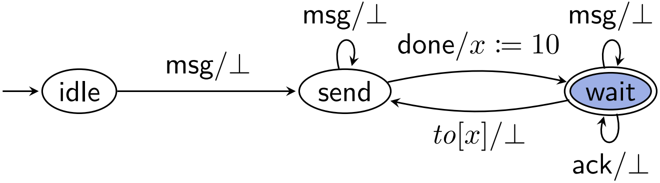


Figure 12: An AwT for an **abstraction** of our network protocol.

$$(\text{idle}, \emptyset) \xrightarrow{\text{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\text{msg}} (\text{send}, \emptyset)$$
$$\xrightarrow{0.5} (\text{send}, \emptyset) \xrightarrow{\text{done}} (\text{wait}, x = 10) \xrightarrow{10} (\text{wait}, x = 0)$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

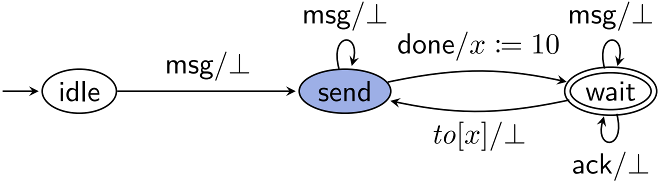


Figure 12: An AwT for an **abstraction** of our network protocol.

$$
\begin{aligned}
(\text{idle}, \emptyset) &\xrightarrow{\mathsf{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\mathsf{msg}} (\text{send}, \emptyset) \\
&\xrightarrow{0.5} (\text{send}, \emptyset) \xrightarrow{\mathsf{done}} (\text{wait}, x = 10) \xrightarrow{10} (\text{wait}, x = 0) \\
&\xrightarrow{to[x]} (\text{send}, \emptyset) \cdots
\end{aligned}
$$

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

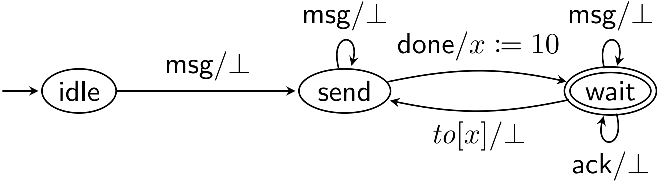


Figure 12: An AwT for an **abstraction** of our network protocol.

$$
\begin{aligned}
(\text{idle}, \emptyset) &\xrightarrow{\mathsf{msg}} (\text{send}, \emptyset) \xrightarrow{1} (\text{send}, \emptyset) \xrightarrow{\mathsf{msg}} (\text{send}, \emptyset) \\
&\xrightarrow{0.5} (\text{send}, \emptyset) \xrightarrow{\mathsf{done}} (\text{wait}, x = 10) \xrightarrow{10} (\text{wait}, x = 0) \\
&\xrightarrow{to[x]} (\text{send}, \emptyset) \cdots
\end{aligned}
$$

So, msg 1 msg 0.5 done 10 $to[x]$ 0 done is accepted by the AwT.

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

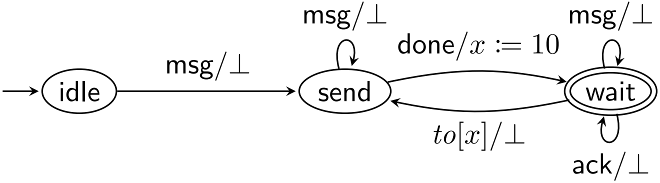


Figure 12: An AwT for an **abstraction** of our network protocol.

We cannot **count**. This is again an **abstraction**.

We must measure **time** and do something once a given amount of time has elapsed.
$\hookrightarrow$ Automata with **timers** (**AwT**, for short).

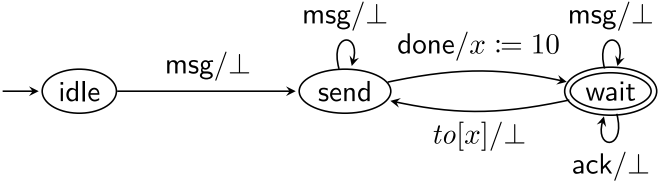


Figure 12: An AwT for an **abstraction** of our network protocol.

We cannot **count**. This is again an **abstraction**.

> **Question.** How to learn an AwT from a system?

Hard task: deducing when a timer is started and at which value.

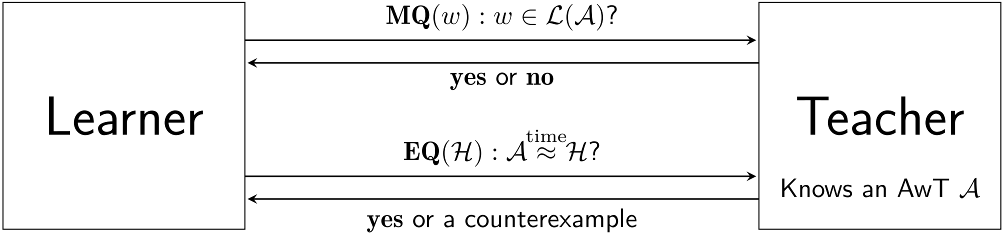


Figure 13: Adaptation of Angluin's framework for AwTs.

Theorem 3. *Let $\mathcal{A}$ be the teacher's **good** AwT and ℓ be the length of the longest counterexample. The $L^{\#}_{\mathsf{MMT}}$ algorithm can learn an AwT equivalent to $\mathcal{A}$ by asking a number of queries **polynomial** in the size of $\mathcal{A}$ and ℓ, and **exponential** in the number of timers of $\mathcal{A}$.*

Main difficulty. Assume a timer x times out exactly at the same time an input i is provided. In which order should the two events be processed?

$$(q_0, x = 0) \xrightarrow{to[x]} (q_1, \perp) \xrightarrow{0} (q_1, \perp) \xrightarrow{i} (q_2, x = 5).$$

$$(q_0, x = 0) \xrightarrow{i} (p_1, x = 0) \xrightarrow{0} (p_1, x = 0) \xrightarrow{to[x]} (p_2, x = 100).$$

Main difficulty. Assume a timer x times out exactly at the same time an input i is provided. In which order should the two events be processed?

$$(q_0, x = 0) \xrightarrow{to[x]} (q_1, \perp) \xrightarrow{0} (q_1, \perp) \xrightarrow{i} (q_2, x = 5).$$

$$(q_0, x = 0) \xrightarrow{i} (p_1, x = 0) \xrightarrow{0} (p_1, x = 0) \xrightarrow{to[x]} (p_2, x = 100).$$

As the learner does **not** know **everything** about the teacher's AwT, some timeouts may be **unexpected**. $\rightsquigarrow$ It is hard to force a specific behavior.

Main difficulty. Assume a timer x times out exactly at the same time an input i is provided. In which order should the two events be processed?

$$(q_0, x = 0) \xrightarrow{to[x]} (q_1, \bot) \xrightarrow{0} (q_1, \bot) \xrightarrow{i} (q_2, x = 5).$$

$$(q_0, x = 0) \xrightarrow{i} (p_1, x = 0) \xrightarrow{0} (p_1, x = 0) \xrightarrow{to[x]} (p_2, x = 100).$$

As the learner does **not** know **everything** about the teacher's AwT, some timeouts may be **unexpected**. $\rightsquigarrow$ It is hard to force a specific behavior.

A **good** AwT ensures that we can force every behavior by carefully choosing the delays.

Problem. Not every AwT is **good**.

Part III – Validating documents

A model checking algorithm for documents in a specific format

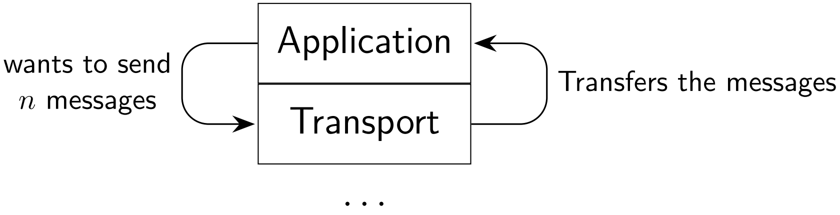


Figure 14: A part of the network stack.

Let us focus on a potential **application** that transmits and receives information as documents following a specific file format.

```
{
  "title": "Active Learning of Automata with Resources",
  "details": {
    "pages": 339,
    "chapters": 11
  },
  "nesting": { "inside": { ... } }
}
```

```
{
  "title": "Active Learning of Automata with Resources",
  "details": {
    "pages": 339,
    "chapters": 11
  },
  "nesting": { "inside": { ... } }
}
```

An **object** is an **unordered** collection of key-value pairs.

There are also arrays (**ordered** collections of values); we mostly ignore them here.

A **JSON document** is composed of **nested** objects and arrays.

```json
{
  "title": "Active Learning of Automata with Resources",
  "details": {
    "pages": 339,
    "chapters": 11
  },
  "nesting": { "inside": { ... } }
}
```

An **object** is an **unordered** collection of key-value pairs.

There are also arrays (**ordered** collections of values); we mostly ignore them here.

A **JSON document** is composed of **nested** objects and arrays.

We want to verify that the document satisfies some **constraints**:

- `"title"` $\mapsto$ string
- `"details"` $\mapsto$ object such that
  - `"pages"` $\mapsto$ integer
  - `"chapters"` $\mapsto$ integer
- And so on.

Classical validation algorithm:

1. Explore the JSON document and the constraints in parallel;
2. If the current value does not match the sub-constraints, stop;
3. Otherwise, repeat recursively.

Classical validation algorithm:

1. Explore the JSON document and the constraints in parallel;
2. If the current value does not match the sub-constraints, stop;
3. Otherwise, repeat recursively.

The constraints can contain Boolean operations.
$\hookrightarrow$ The **same** value must be processed **multiple** times.

Assume we are in a **streaming context**, *e.g.*, by using our network protocol.
$\hookrightarrow$ We receive the document one fragment at a time.

Assume we are in a **streaming context**, *e.g.*, by using our network protocol.
$\hookrightarrow$ We receive the document one fragment at a time.

The classical algorithm must wait for the **whole** document before starting.
$\leadsto$ We waste time.

Our approach is based on **learning** an automaton from the constraints and then use it for **validation**.

Question. Which kind of automaton?

$\hookrightarrow$ A **(visibly) pushdown automaton** (**VPA**), using a **stack**.

Theorem 4. *Let $\mathcal{C}$ be a set of constraints describing JSON documents. There **always** exists a VPA $\mathcal{A}$ whose language is the set of documents that are **valid** for $\mathcal{C}$.*

Theorem 4. *Let $\mathcal{C}$ be a set of constraints describing JSON documents. There* ***always*** *exists a VPA $\mathcal{A}$ whose language is the set of documents that are* ***valid*** *for $\mathcal{C}$.*

Theorem 5 (Isberner, “Foundations of active automata learning: an algorithmic perspective”, 2015). *Let L be a language accepted by some VPA. The TTT_{VPA} algorithm can learn a VPA accepting L by asking a* ***polynomial*** *number of membership and equivalence queries.*

Theorem 4. *Let $\mathcal{C}$ be a set of constraints describing JSON documents. There* ***always*** *exists a VPA $\mathcal{A}$ whose language is the set of documents that are* ***valid*** *for $\mathcal{C}$.*

Theorem 5 (Isberner, "Foundations of active automata learning: an algorithmic perspective", 2015). *Let L be a language accepted by some VPA. The TTT_{VPA} algorithm can learn a VPA accepting L by asking a* ***polynomial*** *number of membership and equivalence queries.*

Problem. There are **exponentially** many permutations of the **(unordered)** keys.
$\leadsto$ There are **exponentially** many valid documents for $\mathcal{C}$.
$\leadsto$ The VPA is **exponential** in the number of keys.

Solution: fix an **order** over the set of keys to get a VPA of reasonable size.

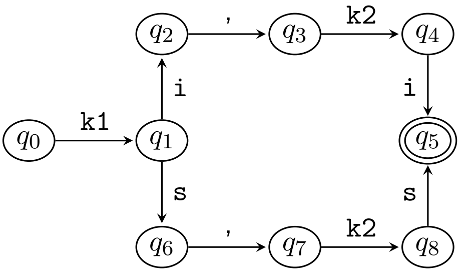
Theorem 6. *Let $\mathcal{C}$ be a set of constraints over the set K of keys and $\mathcal{A}$ be a* ***VPA*** *that recognizes $\mathcal{C}$,* ***with a fixed order*** *over K.*
Then, checking whether a JSON document J satisfies $\mathcal{C}$

- *is in time* ***polynomial*** *in $|J|$ and $|\mathcal{A}|$ and* ***exponential*** *in $|K|$,*
- *and uses an amount of memory* ***polynomial*** *in $|\mathcal{A}|$ and $|K|$.*

Theorem 6. *Let $\mathcal{C}$ be a set of constraints over the set K of keys and $\mathcal{A}$ be a* ***VPA*** *that recognizes $\mathcal{C}$,* ***with a fixed order*** *over K.*
Then, checking whether a JSON document J satisfies $\mathcal{C}$

- *is in time* ***polynomial*** *in $|J|$ and $|\mathcal{A}|$ and* ***exponential*** *in $|K|$,*
- *and uses an amount of memory* ***polynomial*** *in $|\mathcal{A}|$ and $|K|$.*

Main difficulty. Using the VPA to validate JSON documents whose objects do **not** follow the **fixed order**.



Contents of valid objects:

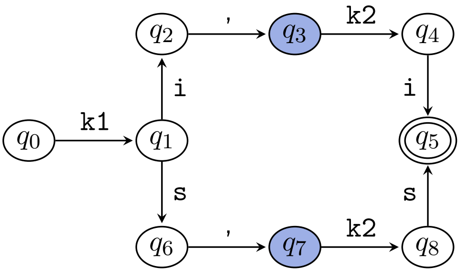
`k1 i , k2 i`

`k2 i , k1 i`

`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.



Contents of valid objects:

`k1 i , k2 i`

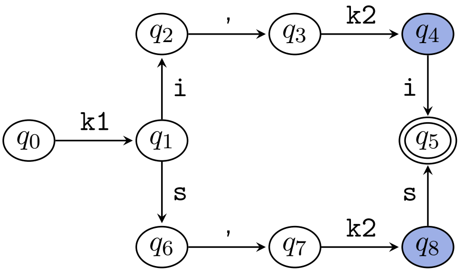
`k2 i , k1 i`

`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.



Contents of valid objects:

`k1 i , k2 i`

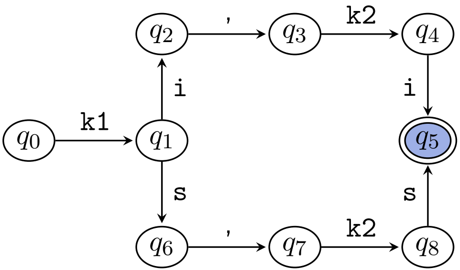
`k2 i , k1 i`

`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.



Contents of valid objects:

`k1 i , k2 i`

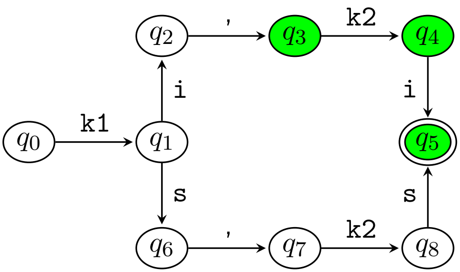
`k2 i , k1 i`

`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.



Contents of valid objects:

`k1 i , k2 i`

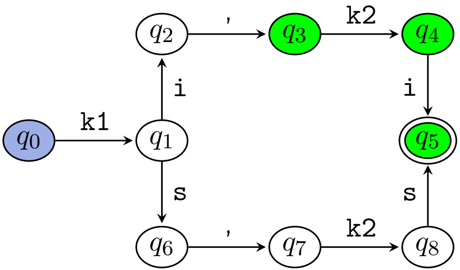
`k2 i , k1 i`

`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.



Contents of valid objects:

`k1 i , k2 i`

`k2 i , k1 i`

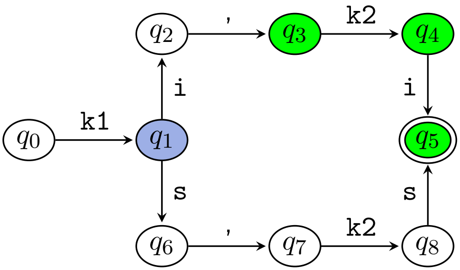
`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.

Potential states for `k1 i`: $\{q_0\}$.



Contents of valid objects:

`k1 i , k2 i`

`k2 i , k1 i`

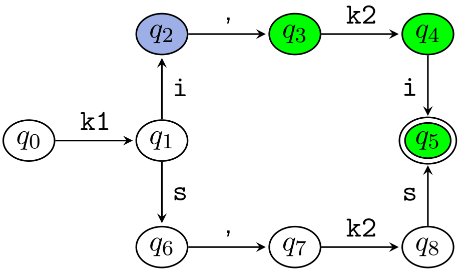
`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.

Potential states for `k1 i`: $\{q_0\}$.
After reading `k1`: $\{q_1\}$.



Contents of valid objects:

`k1 i , k2 i`

`k2 i , k1 i`

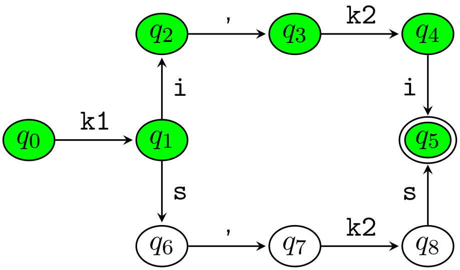
`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.

Potential states for `k1 i`: $\{q_0\}$.
After reading `k1`: $\{q_1\}$.
After reading `k1 i`: $\{q_2\}$.



Contents of valid objects:

`k1 i , k2 i`

`k2 i , k1 i`

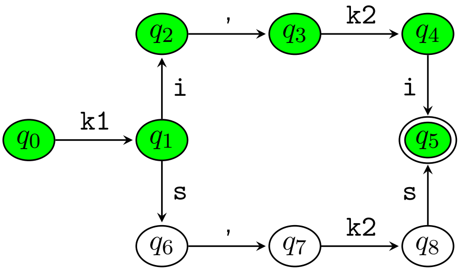
`k1 s , k2 s`

`k2 s , k1 s`

We read `k2 i , k1 i`.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.

Potential states for `k1 i`: $\{q_0\}$.
After reading `k1`: $\{q_1\}$.
After reading `k1 i`: $\{q_2\}$.
Remember $q_0 \xrightarrow{\texttt{k1 i}} q_2$.



Contents of valid objects:

`k1 i , k2 i`
`k2 i , k1 i`
`k1 s , k2 s`
`k2 s , k1 s`

We read `k2 i , k1 i`. $\rightsquigarrow$ **Valid document**.

Potential states for `k2 i`: $\{q_3, q_7\}$.
After reading `k2`: $\{q_4, q_8\}$.
After reading `k2 i`: $\{q_5\}$.
Remember $q_3 \xrightarrow{\texttt{k2 i}} q_5$.

Potential states for `k1 i`: $\{q_0\}$.
After reading `k1`: $\{q_1\}$.
After reading `k1 i`: $\{q_2\}$.
Remember $q_0 \xrightarrow{\texttt{k1 i}} q_2$.

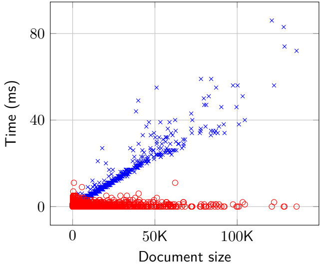
We implemented our algorithm in Java using AutomataLib and LearnLib.[4]

We selected four sets of constraints coming from real-world applications and evaluated the performance on randomly generated JSON documents.

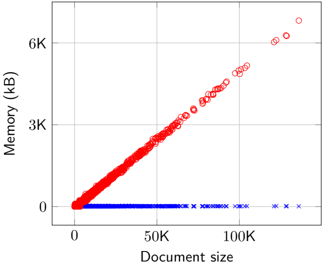
[4] Isberner, Howar, and Steffen, "The Open-Source LearnLib - A Framework for Active Automata Learning", 2015.

Time	Membership	Equivalence	Data structure storage
11305.3 s	4246085.0	36.4	419 kB

(a) Preprocessing (learning + building data structure).



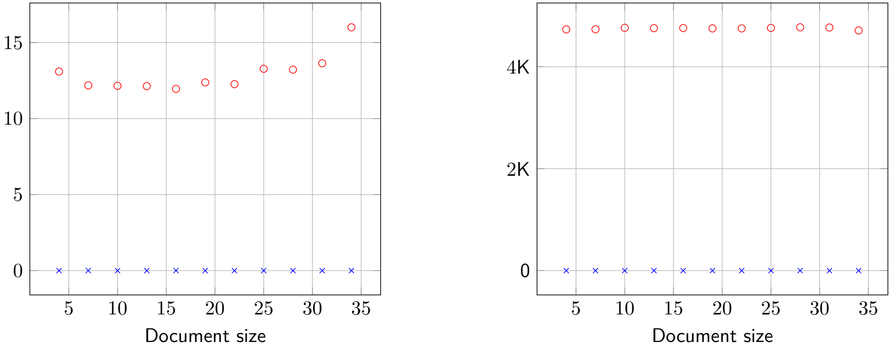
(b) Time usage (ms).



(c) Mem. usage (kB).

Figure 15: Results for **VIM plugins**. $|K| = 16$. Red circles = classical algorithm. Blue crosses = our algorithm.

We use Boolean operations to force the classical algorithm to explore multiple branches, while our algorithm is immediate.



(a) Time usage (ms).

(b) Mem. usage (kB).

Figure 16: Results for a **worst case**. $|K| = 1$. Red circles = classical algorithm. Blue crosses = our algorithm.

Part IV – Conclusion

Active automata learning are used to model check real-world network protocols:

- Ruiter and Poll, "Protocol State Fuzzing of TLS Implementations", 2015;
- Fiterau-Brostean, Janssen, and Vaandrager, "Combining Model Learning and Model Checking to Analyze TCP Implementations", 2016;
- Fiterau-Brostean, Lenaerts, et al., "Model learning and model checking of SSH implementations", 2017.

However, these approaches use "simple" finite automata.
$\rightsquigarrow$ They suffer from the restrictions of FAs.

Goals of the thesis:

- New learning algorithms for automata extended with
  - a **counter** (Bruyère, Pérez, and Staquet, “Learning Realtime One-Counter Automata”, 2022),
  - **timers** (Bruyère, Pérez, Staquet, and Vaandrager, “Automata with Timers”, 2023; Bruyère, Garhewal, et al., “Active Learning of Mealy Machines with Timers”, 2024).
- Model checking algorithm relying on first learning an automaton with a **stack** (Bruyère, Pérez, and Staquet, “Validating Streaming JSON Documents with Learned VPAs”, 2023).

Thank you!

References I

- Angluin, Dana. "Learning Regular Sets from Queries and Counterexamples". In: 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6.
- Bruyère, Véronique, Bharat Garhewal, et al. "Active Learning of Mealy Machines with Timers". In: *CoRR* abs/2403.02019 (2024). DOI: 10.48550/ARXIV.2403.02019. arXiv: 2403.02019.
- Bruyère, Véronique, Guillermo A. Pérez, and Gaëtan Staquet. "Learning Realtime One-Counter Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Springer, 2022, pp. 244–262. DOI: 10.1007/978-3-030-99524-9_13.

References II

- Bruyère, Véronique, Guillermo A. Pérez, and Gaëtan Staquet. "Validating Streaming JSON Documents with Learned VPAs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. Springer, 2023, pp. 271–289. DOI: 10.1007/978-3-031-30823-9_14.
- Bruyère, Véronique, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. "Automata with Timers". In: *Formal Modeling and Analysis of Timed Systems - 21st International Conference, FORMATS 2023, Antwerp, Belgium, September 19-21, 2023, Proceedings*. Ed. by Laure Petrucci and Jeremy Sproston. Vol. 14138. Springer, 2023, pp. 33–49. DOI: 10.1007/978-3-031-42626-1_3.

References III

- Fiterau-Brostean, Paul, Ramon Janssen, and Frits W. Vaandrager. "Combining Model Learning and Model Checking to Analyze TCP Implementations". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Springer, 2016, pp. 454–471. DOI: 10.1007/978-3-319-41540-6_25.
- Fiterau-Brostean, Paul, Toon Lenaerts, et al. "Model learning and model checking of SSH implementations". In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*. Ed. by Hakan Erdogmus and Klaus Havelund. ACM, 2017, pp. 142–151. DOI: 10.1145/3092282.3092289.

References IV

- Isberner, Malte. "Foundations of active automata learning: an algorithmic perspective". PhD thesis. Technical University Dortmund, Germany, 2015. URL: https://hdl.handle.net/2003/34282.
- Isberner, Malte, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib - A Framework for Active Automata Learning". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Springer, 2015, pp. 487–495. DOI: 10.1007/978-3-319-21690-4_32.

References V

Ruiter, Joeri de and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.