Benchmarks for Automata Learning and Conformance Testing

Daniel Neider[1], Rick Smetsers[2⋆], Frits Vaandrager[2⋆⋆], and Harco Kuppens[2]

[1] Max Planck Institute for Software Systems, Kaiserslautern
[2] Institute for Computing and Information Sciences, Radboud University, Nijmegen

Abstract. We describe a large collection of benchmarks, publicly available through the wiki `automata.cs.ru.nl`, of different types of state machine models: DFAs, Moore machines, Mealy machines, interface automata and register automata. Our repository includes both randomly generated state machines and models of real protocols and embedded software/hardware systems. These benchmarks will allow researchers to evaluate the performance of new algorithms and tools for active automata learning and conformance testing.

1 Introduction

Active automata learning (or model learning) aims to construct black-box state machine models of software and hardware systems by providing inputs and observing outputs. State machines are crucial for understanding the behavior of many software systems, such as network protocols and embedded control software, as they allow us to reason about communication errors and component compatibility. Model learning is emerging as a highly effective bug-finding technique, and is slowly becoming a standard tool in the toolbox of the software engineer [68, 35]. Bernhard Steffen has been (and still is) the main intellectual driving force behind this important development, and together with his students and coworkers he has made numerous important contributions to the theory and application of model learning, see e.g., [9, 14, 15, 34, 35, 37, 38, 63]. His ideas have been implemented in the open source automata learning framework LearnLib [55, 56, 49], which has become the most prominent tool in this area.

Many model learning algorithms have been proposed in the literature, for instance by Angluin [8], Rivest & Schapire [57], Kearns & Vazirani [40], Shahbaz & Groz [61], Bollig et al. [10], Howar [33], Isberner et al. [37], Aarts et al. [1], Cassel et al. [14, 15], and Moerman et al. [50]. Often variations of algorithms exist for different classes of models, e.g., DFAs, Mealy machines, Moore machine, interface automata, and various forms of register automata. Active automata learning is closely related to conformance testing [9]. Whereas automata learning aims at constructing hypothesis models from observations, conformance

⋆ Supported by NWO/EW project 628.001.009 (LEMMA).
⋆⋆ Supported by NWO project 13859 (SUMBAT).

testing checks whether a system under test conforms to a given model. Conformance test tools play a crucial role within active automata learning, as a way to determine whether a hypothesis model is correct or not. Also in the literature on conformance testing many algorithms have been proposed for different model classes, for surveys see [43, 44, 67, 23].

Although there has been some experimental work on evaluating algorithms for model learning and conformance testing, see e.g., [3, 12, 23, 24], the number of realistic benchmarks is rather limited, and different papers use different models and/or black-box implementations. Often the benchmarks used are small, academic, or randomly generated. Small, academic benchmarks are useful during tool development, but do not say much about the performance on industrial cases. The performance of algorithms on randomly generated benchmarks is often radically different from performance on benchmarks based on real systems that occur in practice. A mature field is characterized by the presence of a rich set of shared benchmarks, used to evaluate the efficiency of algorithms and tools, and as challenges for pushing the state-of-the-art.

In this article, we describe a large collection of benchmarks, publicly available through the wiki repository `automata.cs.ru.nl`, that includes both randomly generated state machines and models of real protocols and embedded software/hardware systems. Our benchmarks will allow researchers to compare the performance of algorithms and tools for learning and conformance testing, to check whether tools and methods advance, and to demonstrate that new methods are effective.

We are aware of a few other repositories with benchmarks for model learning and/or conformance testing. The ACM/SIGDA benchmark dataset [12, 24] contains behavioral models for testing, logic synthesis and optimization of circuits. We have included Mealy machine versions of these benchmarks in our repository. The goal of the GitHub repository AutomatArk [19] is to collect benchmark problems for different models of automata, transducers, and related logics. In particular, AutomatArk contains NFAs that are adapted from a few verification case studies. The RERS challenges [38], `www.rers-challenge.org`, aim to provide realistic benchmarks that allow researchers to compare different software validation techniques, e.g., static analysis, model checking, symbolic execution and (model-based) testing. Benchmarks of previous challenges are still available via the website. The StaMinA competition [71], `stamina.chefbe.net`, focused on the complexity of learning with respect to the alphabet size. The competition is closed, but the website still hosts all of its benchmarks, a total of 100. Finally, we mention the Very Large Transition Systems (VLTS) benchmark suite, `http://cadp.inria.fr/resources/vlts/`, which has been set up by CWI and INRIA to support the evaluation of algorithms and tools for explicit state verification. Whereas the benchmarks in our repository model the behavior of individual components with at most a few thousand states, the VLTS benchmarks typically describe the behavior of concurrent systems that are composed of multiple components and that have a global state space with milions of states.

Most VLTS benchmarks are completely out of reach for state-of-the-art learning and testing tools.

The remainder of this article is organized as follows. In Section 2, we discuss the different types of automata frameworks that are supported in our repository (DFAs, Moore machines, Mealy machines, interface automata, and register automata) and behavior preserving translations between these frameworks. Even though most of the definitions are standard, and most of the translations are folklore, this is the first time all these definitions and translations are presented together in a comprehensive manner, using consistent terminology and notation. The translations play a crucial role in our automata repository, since they allow us to transfer benchmarks from one framework to another, and thus obtain many benchmarks "for free". Section 3 gives an overview of the network protocols, embedded controllers, circuits, and other realistic applications for which models have been included in our benchmark collection. Section 4 discusses algorithms for generating the random automaton models that we have included in our repository. Finally, Section 5 draws some conclusions.

2 State Machine Frameworks

Below we recall the definitions of the different types of state machines for which we have collected benchmarks, discuss data formats to represent different model classes, define the corresponding notions of behavioral equivalence, and describe behavior preserving translations between types of state machines.

Figure 1 presents an overview of the different state machine frameworks that we will discuss, and their relationships. For the finite state frameworks indicated with red boxes, benchmark models have been included in our repository: *DFAs*, *Moore machines*, *Mealy machines*, *deterministic finite interface automata (DFIAs)*, and *register automata*. For some frameworks, more general (nondeterministic and infinite state) variants have been studied in the literature: *general-*

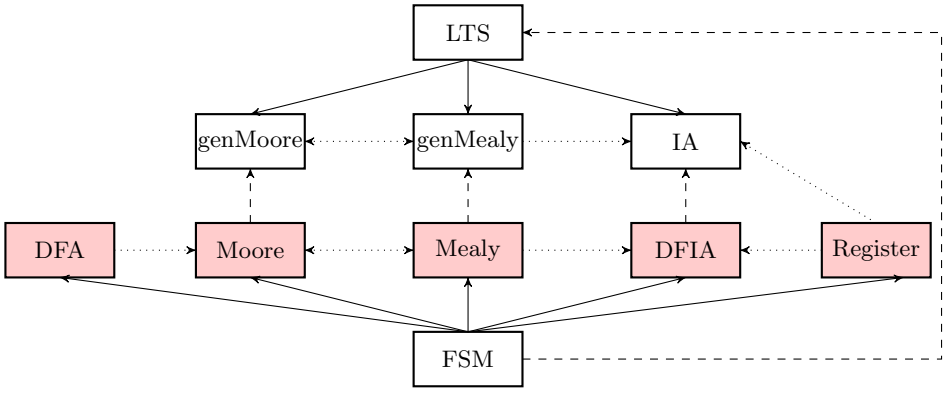


Fig. 1: Overview of state machine frameworks.

ized Moore machines, *generalized Mealy machines*, and *interface automata (IAs)*. All finite state frameworks have an underlying *finite state machine (FSM)*, and all infinite state frameworks have an underlying *labeled transition system (LTS)*. In Figure 1, a regular arrow indicates that one framework is a substructure of another, a dashed arrow that one framework is a special case of another, and a dotted arrow that a behavior preserving translation exists.

2.1 Labeled transition systems

All the state machines that we consider are labeled, directed graphs, equipped with some extra structure. Following standard terminology, we refer to the underlying graphs as *labeled transition systems* [41].

Definition 1 (Labeled transition systems). *A* labeled transition system (LTS) *is a tuple* $\mathcal{S} = \langle Q, Q_0, A, \rightarrow \rangle$*, where*

- Q *is a non-empty set of* states*,*
- $Q_0 \subseteq Q$ *is a non-empty set of* initial states*,*
- A *is a set of* actions*, and*
- $\rightarrow \subseteq Q \times A \times Q$ *is a* transition relation*.*

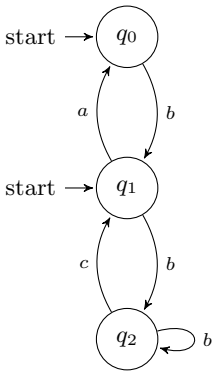
We write $q \xrightarrow{a} q'$ *if* $(q, a, q') \in \rightarrow$*. An LTS* $\mathcal{S}$ *is* deterministic *if* Q_0 *is a singleton set, and for each state* $q \in Q$ *and each action* $a \in A$*, there is at most one state* $q' \in Q$ *such that* $q \xrightarrow{a} q'$*. An action* $a \in A$ *is* enabled *in state* $q \in Q$*, notation* $q \xrightarrow{a}$*, if there exists a state* $q' \in Q$ *such that* $q \xrightarrow{a} q'$*. An LTS* $\mathcal{S}$ *is* completely specified *(or* complete*) if each action is enabled in each state. An LTS* $\mathcal{S}$ *is* finite *and is called a* finite-state machine (FSM) *if sets* Q *and* $\rightarrow$ *are both finite.*

For a sequence of actions $\sigma = a_1 a_2 \cdots a_m \in A^*$ *and states* $q, q' \in Q$*, we write* $q \stackrel{\sigma}{\Rightarrow} q'$ *if there exist states* $q_0, \ldots, q_m \in S$ *such that* $q_0 = q$*,* $q_m = q'$*, and* $q_{j-1} \xrightarrow{a_j} q_j$ *for all* $1 \leq j \leq m$*.*

FSMs and the various extensions that we will review below are syntactically represented in our repository using the graph description language DOT [28]. Scripts are provided to translate between DOT and other common formats for representing state machines. Figure 2 shows the graphical representation of a simple FSM (left) and its representation in DOT (right). The graphical representation follows the usual conventions for representing graphs. Initial states are indicated by a small incoming edge. The DOT representation first lists all the states, then the start states, and then the transitions. In order to mark the initial states, an auxiliary "invisible" node is created with edges to all the start states. Actions are indicated as labels of transitions.

Definition 2 (Bisimulation). *Let* $\mathcal{S}_1 = \langle Q_1, Q_0^1, A, \rightarrow_1 \rangle$*,* $\mathcal{S}_2 = \langle Q_2, Q_0^2, A, \rightarrow_2 \rangle$ *be LTSs. A* bisimulation *between* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *is a relation* $R \subseteq Q_1 \times Q_2$ *that satisfies:*

1. *for every* $q_1 \in Q_0^1$ *there exists a* $q_2 \in Q_0^2$ *such that* $(q_1, q_2) \in R$*,*




```
digraph S {
__start0 [label="" shape="none"]
__start1 [label="" shape="none"]
q0 [shape="circle" label="s0"]
q1 [shape="circle" label="s1"]
q2 [shape="circle" label="s2"]
__start0 -> q0
__start1 -> q1
q0 -> q1 [label="b"]
q1 -> q0 [label="a"]
q1 -> q2 [label="b"]
q2 -> q1 [label="c"]
q2 -> q2 [label="b"]
}
```

Fig. 2: An FSM and its representation in DOT.

2. *for every* $q_2 \in Q_0^2$ *there exists a* $q_1 \in Q_0^1$ *such that* $(q_1, q_2) \in R$,
3. *for every* $q_1, q_1' \in Q_1$, $a \in A$ *and* $q_2 \in Q_2$ *with* $(q_1, q_2) \in R$ *and* $q_1 \xrightarrow{a} q_1'$, *there exists a* $q_2' \in Q_2$ *such that* $q_2 \xrightarrow{a} q_2'$ *and* $(q_1', q_2') \in R$,
4. *for every* $q_2, q_2' \in Q_2$, $a \in A$ *and* $q_1 \in Q_1$ *with* $(q_1, q_2) \in R$ *and* $q_2 \xrightarrow{a} q_2'$, *there exists a* $q_1' \in Q_1$ *such that* $q_1 \xrightarrow{a} q_1'$ *and* $(q_1', q_2') \in R$.

We say that $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are* bisimilar*, and write* $\mathcal{S}_1 \simeq \mathcal{S}_2$*, if there exists a bisimulation between* $\mathcal{S}_1$ *and* $\mathcal{S}_2$.

2.2 Finite automata

A finite automaton [32] extends an FSM by identifying some states as accepting.

Definition 3 (Finite automaton). *A* (nondeterministic) finite automaton (or NFA) *is a tuple* $\mathcal{A} = \langle Q, Q_0, \Sigma, \to, F \rangle$*, where* $\langle Q, Q_0, \Sigma, \to \rangle$ *is an FSM and* $F \subseteq Q$ *is a set of* final *(or* accepting*) states. Elements of* Σ *are referred to as* input symbols*. A* deterministic finite automataton (DFA) *is an NFA for which the underlying FSM is deterministic and complete.*

In the DOT format, accepting states of a finite automaton are denoted by a double circle, following the standard convention:

```
digraph g {
  ...
  q [shape="doublecircle"]
  ...
}
```

Definition 4 (Equivalence of NFAs). *A finite sequence (or* word*)* $w \in \Sigma^*$ *is* accepted *by NFA* $\mathcal{A}$ *iff there exists an initial state* $q \in Q_0$ *and a final state* $q' \in F$ *such that* $q \overset{w}{\Rightarrow} q'$*. If* w *is not accepted then we say it is* rejected*. The*

language $L(\mathcal{A})$ *of* $\mathcal{A}$ *is the set of all words accepted by* $\mathcal{A}$*. Two NFAs* $\mathcal{A}$ *and* $\mathcal{B}$ *are* equivalent*, notation* $\mathcal{A} \approx \mathcal{B}$*, if they have the same set of input symbols and* $L(\mathcal{A}) = L(\mathcal{B})$*.*

2.3 Moore machines

A (generalized) Moore machine [51] extends an LTS by assigning an output to each state.

Definition 5 (Generalized Moore machine). *A* generalized Moore machine (or genMoore) *is a tuple* $\mathcal{M} = \langle Q, Q_0, \Sigma, \Gamma, \to, \omega \rangle$*, where* $\langle Q, Q_0, \Sigma, \to \rangle$ *is an LTS,* Γ *is a set of* output symbols*, and* $\omega : Q \to \Gamma$ *is an* output function*. We call elements of* Σ input symbols*. A* Moore machine *is a genMoore for which the underlying LTS is deterministic, complete and finite.*

In the DOT representation of a Moore machine, the value o of the output function in state q is listed after a "|" in the label of state q:

```
digraph g {
  ...
  q [shape="record", style="rounded", label="{ q | o }"]
  ...
}
```

Definition 6 (Equivalence of genMoores). *Suppose* $w = i_1 i_2 \cdots i_m \in \Sigma^*$*,* $q_0 \in Q_0$*, and* $q_1, \ldots, q_m \in Q$ *with* $q_{j-1} \xrightarrow{i_j} q_j$ *for all* $1 \leq j \leq m$*. Then the sequence* $\omega(q_1) \cdots \omega(q_m) \in \Gamma^*$ *is an* output of genMoore $\mathcal{M}$ in reponse to w.[3] *The* output function *of* $\mathcal{M}$ *is the function* $\lambda_{\mathcal{M}}$ *that assigns to each input word* $w \in \Sigma^*$ *the set of all outputs of* $\mathcal{M}$ *in response to* w*. Two genMoores* $\mathcal{M}$ *and* $\mathcal{N}$ *are* equivalent*, notation* $\mathcal{M} \approx \mathcal{N}$*, if they have the same input symbols and* $\lambda_{\mathcal{M}} = \lambda_{\mathcal{N}}$*.*

A DFA $\mathcal{A} = \langle Q, Q_0, \Sigma, \to, F \rangle$ can be translated to a Moore machine $\mathsf{DFA2Moore}(\mathcal{A}) = \langle Q, Q_0, \Sigma, \Gamma, \to, \omega \rangle$ by associating to each state $q \in Q$ an output that indicates whether or not q is final [32]. That is, we define $\Gamma = \{0, 1\}$ and

$$\omega(q) = \begin{cases} 1 \text{ if } q \in F, \\ 0 \text{ otherwise.} \end{cases}$$

Suppose $\mathcal{A}$ and $\mathcal{B}$ are DFAs with $\epsilon \in L(\mathcal{A}) \Leftrightarrow \epsilon \in L(\mathcal{B})$. Then $\mathcal{A} \approx \mathcal{B}$ iff $\mathsf{DFA2Moore}(\mathcal{A}) \approx \mathsf{DFA2Moore}(\mathcal{B})$. Thus, the translation $\mathsf{DFA2Moore}$ preserves the behavior of DFAs. The counterexample of Figure 3 shows that if we lift translation Moore to NFAs, the behavior is no longer preserved: $\mathcal{A} \approx \mathcal{B}$ since $L(\mathcal{A}) = L(\mathcal{B}) = \{a, aa\}$, but $\mathsf{DFA2Moore}(\mathcal{A}) \not\approx \mathsf{DFA2Moore}(\mathcal{B})$ since $\lambda_{\mathsf{DFA2Moore}(\mathcal{A})}(a) = \{0, 1\}$ and $\lambda_{\mathsf{DFA2Moore}(\mathcal{B})}(a) = \{1\}$.

[3] Following Hopcroft & Ullman [32], we ignore the initial output in order to obtain equivalence of Moore and Mealy machines.

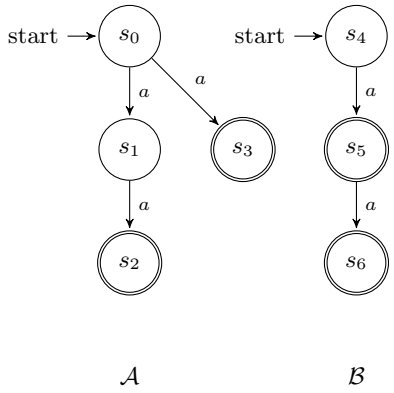


Fig. 3: Two NFAs $\mathcal{A}$ and $\mathcal{B}$ with $\mathcal{A} \approx \mathcal{B}$ and $\mathsf{DFA2Moore}(\mathcal{A}) \not\approx \mathsf{DFA2Moore}(\mathcal{B})$.

2.4 Mealy machines

A (generalized) Mealy machine [48] is an LTS in which the labels of transitions are input/output pairs.

Definition 7 (Generalized Mealy machine). *A* generalized Mealy machine (genMealy) *is a tuple $\mathcal{M} = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow \rangle$, where $\langle Q, Q_0, \Sigma \times \Gamma, \rightarrow \rangle$ is an LTS. We refer to elements of Σ as* input symbols *and to elements of Γ as* output symbols. *We write $q \xrightarrow{i/o} q'$ if $(q, (i, o), q') \in \rightarrow$. We say that $\mathcal{M}$ is* input enabled *if, for each state q and input symbol i, there exists an output symbol o and a state q' such that $q \xrightarrow{i/o} q'$. We call $\mathcal{M}$* deterministic *if Q_0 is a singleton set, and for each state q and each input i, there is exactly one output o and one state q' such that $q \xrightarrow{i/o} q'$. We call $\mathcal{M}$* finite *if its underlying LTS is finite, and a* Mealy machine *if it is input enabled, deterministic, and finite.*

In the DOT encoding of a Mealy machine, inputs and outputs are separated by a "/" in the definition of transitions:

```
digraph g {
  ...
  q1 -> q2 [label="i/o"]
  ...
}
```

Definition 8 (Equivalence of genMealys). *Suppose $w = i_1 i_2 \cdots i_m \in \Sigma^*$ and $u = o_1 o_2 \cdots o_m \in \Gamma^*$. Then u is an* output *of genMealy $\mathcal{M}$ in reponse to w if there exists $q \in Q_0$ and $q' \in Q$ such that $q \stackrel{z}{\Rightarrow} q'$, where $z = (i_1, o_1)(i_2, o_2) \cdots (i_m, o_m)$. The* output function *$\lambda_{\mathcal{M}}$ of $\mathcal{M}$ assigns to each input word $w \in \Sigma^*$ the set of outputs of $\mathcal{M}$ in response to w. Generalized Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are* equivalent*, notation $\mathcal{M} \approx \mathcal{N}$, if they have the same input symbols and $\lambda_{\mathcal{M}} = \lambda_{\mathcal{N}}$.*

Equivalence of deterministic genMealys can alternatively be characterized using bisimulations. Call genMealy's $\mathcal{M}$ and $\mathcal{N}$ *bisimilar*, written $\mathcal{M} \simeq \mathcal{N}$, if they have the same input symbols and their underlying LTSs are bisimilar. Then the following proposition holds:

Proposition 1. *Let $\mathcal{M}$ and $\mathcal{N}$ be deterministic genMealys. Then $\mathcal{M} \approx \mathcal{N}$ iff $\mathcal{M} \simeq \mathcal{N}$.*

Each generalized Moore machine $\mathcal{M} = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow, \omega \rangle$ can be translated to a generalized Mealy machine $\mathsf{Moore2Mealy}(\mathcal{M}) = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow' \rangle$ by moving the output symbol of each state to all of the incoming transitions of that state. Thus, for each transition $q \xrightarrow{i} q'$ of $\mathcal{M}$, $\mathsf{Moore2Mealy}(\mathcal{M})$ has a transition $q \xrightarrow{i/\omega(q')} q'$. Then we have $\lambda_{\mathcal{M}} = \lambda_{\mathsf{Moore2Mealy}(\mathcal{M})}$ (see e.g., [32]). This implies that genMoores $\mathcal{M}$ and $\mathcal{N}$ are equivalent iff $\mathsf{Moore2Mealy}(\mathcal{M})$ and $\mathsf{Moore2Mealy}(\mathcal{N})$ are equivalent. The reader may check that if we take a Moore machine and apply translation $\mathsf{Moore2Mealy}$, the result is a Mealy machine.

Example 1. Figure 4 shows a Moore machine and its associated Mealy machine.

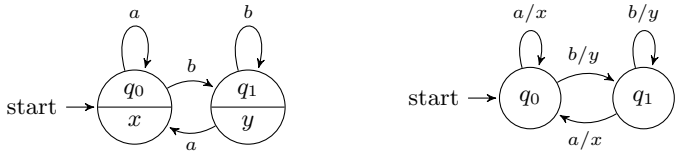


Fig. 4: A Moore machine (left) and its translation to a Mealy machine (right).

Conversely, a generalized Mealy machine $\mathcal{M} = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow \rangle$ can be translated to a generalized Moore machine $\mathsf{Mealy2Moore}(\mathcal{M}) = \langle Q', Q'_0, \Sigma, \Gamma, \rightarrow', \omega \rangle$ by taking the output of a state to be equal to the output of the preceding transition. For initial states we pick an arbitrary output $o_0 \in \Gamma$. Formally:

- $Q' = \Gamma \times Q$,
- $Q'_0 = \{(o_0, q) \mid q \in Q_0\}$, where o_0 is an arbitrarily element of Γ,[4]
- $\rightarrow'$ is the smallest set such that $o \in \Gamma$ and $q \xrightarrow{i/o'} q'$ implies $(o, q) \xrightarrow{i} (o', q')$,
- $\omega((o, q)) = o$.

Then we have $\lambda_{\mathcal{M}} = \lambda_{\mathsf{Mealy2Moore}(\mathcal{M})}$ (see e.g., [32]). This implies that generalized Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are equivalent iff $\mathsf{Mealy2Moore}(\mathcal{M})$ and $\mathsf{Mealy2Moore}(\mathcal{N})$ are equivalent. The reader may check that if we take a Mealy machine and apply translation $\mathsf{Mealy2Moore}$, the result is a Moore machine.

Example 2. Figure 5 shows a Mealy machine and its associated Moore machine.

[4] If $\Gamma = \emptyset$ then also $\rightarrow = \emptyset$, which means that $\mathcal{M}$ is equivalent to $\mathcal{M}$ with Γ replaced by an arbitrary set. Thus, we may assume w.l.o.g. that $\Gamma \neq \emptyset$.

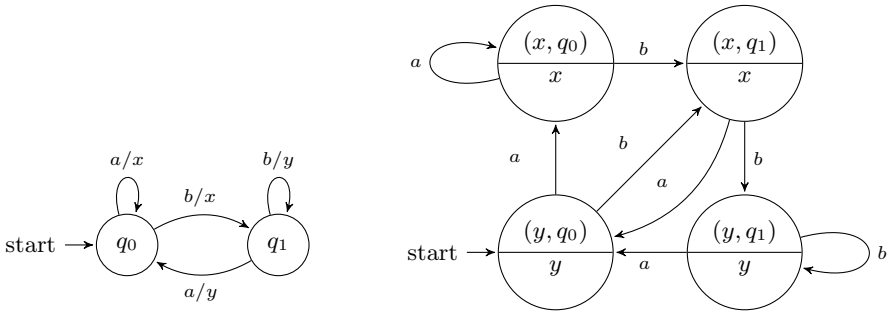


Fig. 5: A Mealy machine (left) and its translation to a Moore machine (right).

2.5 Interface automata

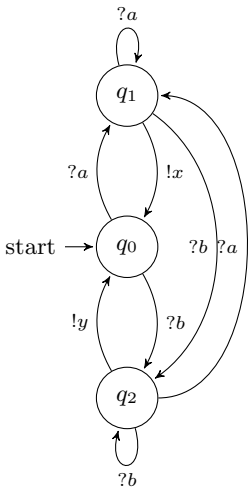
A restriction of Mealy and Moore machines is that each input generates exactly one output. In real-world systems, some inputs do not induce any output, whereas others induce several consecutive outputs. In order to model such behaviors, De Alfaro and Henzinger [21] introduced interface automata, a modeling framework related to the I/O automata of Lynch & Tuttle [47, 46] and Jonsson [39], and the I/O transition systems of Tretmans [65, 66]. Interface automata extend LTSs by declaring actions to be either inputs or outputs.

Definition 9 (Interface automata). *An* interface automaton (IA) *is a tuple* $\mathcal{T} = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow \rangle$*, where* $\langle Q, Q_0, \Sigma \cup \Gamma, \rightarrow \rangle$ *is an LTS and* $\Sigma \cap \Gamma = \emptyset$*. We refer to elements of* Σ *as* input symbols *and to elements of* Γ *as* output symbols*. An interface automaton is* deterministic *(resp.* finite*) if its underlying LTS is deterministic (resp. finite). We refer to a finite deterministic interface automaton as a* DFIA*.*

Figure 6 shows the graphical representation of a simple DFIA (left) and its representation in DOT (right). The DFIA has inputs $\Sigma = \{a, b\}$ and outputs $\Gamma = \{x, y\}$. There are three states: an initial idle state q_0, a state q_1 in which output x is produced, and a state q_2 in which output y is produced. From each state, input a brings the DFIA to state q_1 and input b brings it to state q_2. In DOT format, input symbols of an IA are of the form `?a`, whereas output symbols are of the form `!x`.

Various preorders have been advocated for IAs: Lynch and Tuttle propose inclusion of (fair) traces [47], De Alfaro and Henzinger alternating refinement [21], Tretmans [65] the IOCO conformance relation, and Volpato and Tretmans [69] UIOCO conformance. For deterministic automata all these relations coincide, and their kernel coincides with bisimulation equivalence. Therefore, since our benchmark repository focuses on deterministic IAs, we only consider bisimulation as behavioral equivalence on IAs.

Definition 10 (Equivalence of IAs). *Interface automata* $\mathcal{T}$ *and* $\mathcal{U}$ *are* bisimilar*, written* $\mathcal{T} \simeq \mathcal{U}$*, if they have the same input symbols and their underlying LTSs are bisimilar.*




```
digraph S {
__start0 [label="" shape="none"]
q0 [shape="circle" label="q0"]
q1 [shape="circle" label="q1"]
q2 [shape="circle" label="q2"]
__start0 -> q0
q0 -> q1 [label="?a"]
q0 -> q2 [label="?b"]
q1 -> q0 [label="!x"]
q1 -> q1 [label="?a"]
q1 -> q2 [label="?b"]
q2 -> q0 [label="!y"]
q2 -> q1 [label="?a"]
q2 -> q2 [label="?b"]
}
```

Fig. 6: A DFIA and its representation in DOT.

Suppose $\mathcal{M} = \langle Q, Q_0, \Sigma, \Gamma, \rightarrow \rangle$ is a generalized Mealy machine with disjoint input and output symbols. Then $\mathcal{M}$ can be translated to an interface automaton $\mathsf{Mealy2IA}(\mathcal{M})$ by adding states $\Gamma \times Q$, and splitting each transition $q \xrightarrow{i/o} q'$ of $\mathcal{M}$ into a pair of consecutive transitions $q \xrightarrow{i} (o, q')$ and $(o, q') \xrightarrow{o} q'$. Note that if $\mathcal{M}$ is a Mealy machine, $\mathsf{Mealy2IA}(\mathcal{M})$ is a DFIA. Figure 7 shows a Mealy machine and its associated DFIA.

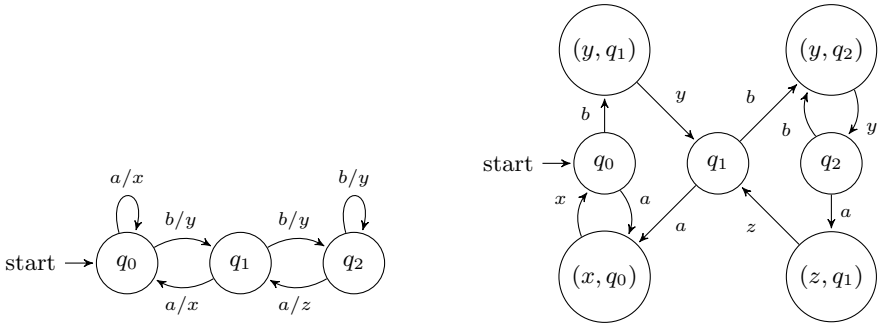


Fig. 7: A Mealy machine (left) and its translation to a DFIA (right).

Proposition 2. *Let $\mathcal{M}$ and $\mathcal{N}$ be deterministic genMealys. Then $\mathcal{M} \approx \mathcal{N}$ iff $\mathsf{Mealy2IA}(\mathcal{M}) \simeq \mathsf{Mealy2IA}(\mathcal{N})$.*

For any generalized Mealy machine $\mathcal{M}$, $\mathsf{Mealy2IA}(\mathcal{M})$ has a specific form in which inputs and outputs alternate: (a) the set of states can be partitioned

into two sets Q_{in} and Q_{out}, with $Q_0 \subseteq Q_{\mathsf{in}}$, (b) states in Q_{in} enable no outputs, whereas states in Q_{out} enable no inputs, (c) all transitions go from states in Q_{in} to states in Q_{out}, or from states in Q_{out} to states in Q_{in} (i.e., the underlying graph is bipartite). We call an IA $\mathcal{T}$ that satisfies properties (a)-(c) *Mealy-like*. Any Mealy-like IA $\mathcal{T}$ can be translated to a generalized Mealy machine $\mathsf{IA2Mealy}(\mathcal{T})$ by taking Q_{in} as set of states, Q_{in} as the set of initial states, and merging each pair of consecutive transitions $q \xrightarrow{i} q' \xrightarrow{o} q''$ of $\mathcal{T}$ into a single transition $q \xrightarrow{i/o} q''$. Note that if $\mathcal{T}$ is a Mealy-like DFIA with each input enabled in each state from Q_{in} and a single output enabled in each state of Q_{out}, $\mathsf{IA2Mealy}(\mathcal{T})$ is a Mealy machine. Also note that $\mathsf{IA2Mealy} \circ \mathsf{Mealy2IA}$ is the identity function, whereas $\mathsf{Mealy2IA} \circ \mathsf{IA2Mealy}$ is not. However, we do have the following proposition:

Proposition 3. *Let $\mathcal{T}$ and $\mathcal{U}$ be Mealy-like deterministic IAs. Then $\mathcal{T} \simeq \mathcal{U}$ iff $\mathsf{IA2Mealy}(\mathcal{T}) \approx \mathsf{IA2Mealy}(\mathcal{U})$.*

2.6 Register automata

Register automata extend FSMs with data values that may be communicated, stored and tested. Below we recall the definition of register automata from [15], slightly adapted to the setting of interface automata. Register automata are parameterized on a vocabulary that determines how data can be tested, which in our setting is called a structure.[5] A *(relational) structure* is a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is an unbounded domain of *data values*, and $\mathcal{R}$ is a collection of *relations* on $\mathcal{D}$. Relations in $\mathcal{R}$ can have arbitrary arity. Known constants can be represented by unary relations. Examples of simple structures include:

- $\langle \mathbb{N}, \{=\} \rangle$, the natural numbers with equality; instead of the set of natural numbers, we could consider any other unbounded domain, e.g., the set of strings (representing passwords or usernames).
- $\langle \mathbb{R}, \{<\} \rangle$, the real numbers with inequality: this structure also allows one to express equality between elements.

Operations, such as increments, addition and subtraction, can in this framework be represented by relations. For instance, addition can be represented by a ternary relation $p_1 = p_2 + p_3$. In the following definitions, we assume that some structure $\langle \mathcal{D}, \mathcal{R} \rangle$ has been fixed.

We assume a set of *registers* $\mathcal{V} = \{x_1, x_2, \ldots\}$, and we assume that actions carry a single formal data parameter $p \notin \mathcal{V}$.[6] A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) over the formal parameter p and the registers. We use Φ to denote the set of guards. An *assignment* is a partial function in $\mathcal{V} \rightharpoonup (\mathcal{V} \cup \{p\})$. We use Υ to denote the set of assignments. A *valuation* is a partial function in $(\mathcal{V} \cup \{p\}) \rightharpoonup \mathcal{D}$.

[5] In [15] this is called a *theory*, but we prefer the standard terminology from logic [18].

[6] Actually, our repository supports actions with zero or more data parameters, but this assumption simplifies the presentation.

Definition 11 (Register automaton). *A* register automaton *(RA) is a tuple* $\mathcal{A} = \langle L, L_0, \mathcal{X}, \Sigma, \Gamma, \rightarrow \rangle$*, where* $\langle L, L_0, (\Sigma \cup \Gamma) \times \Phi \times \Upsilon, \rightarrow \rangle$ *is an FSM, we refer to elements of* L *as* locations*,* $\Sigma \cap \Gamma = \emptyset$*,* $\mathcal{X}$ *maps each location* $l \in L$ *to a finite set* $\mathcal{X}(l)$ *of registers, and for each transition* $\langle l, a, g, \pi, l' \rangle \in \rightarrow$*,* g *is a guard over* $\mathcal{X}(l) \cup \{p\}$ *and* π *is a mapping from* $\mathcal{X}(l')$ *to* $\mathcal{X}(l) \cup \{p\}$*. Function* π *specifies, for each register* x *from target state* l'*, the parameter or register from source state* l *whose value will be assigned to* x*.*

Within the Tomte and RALib tools, XML formats have been defined for representing register automata syntactically. We will not discuss these formats here but refer to the tool websites `http://tomte.cs.ru.nl/` and `https://bitbucket.org/learnlib/ralib/` for more details.

Example 3. Figure 8 shows a register automaton over structure $\langle \mathbb{N}, \{=\} \rangle$ that models a FIFO-set with capacity two, similar to an example in [34]. A FIFO-set is a queue in which only different values can be stored. The automaton has an input Push that tries to insert a value in the queue, and an input Pop that tries to retrieve a value from the queue. Push triggers an output NOK if the input value is already in the queue or if the queue is full. Pop triggers an output NOK if the queue is empty, and otherwise an output Out with as parameter the oldest value from the queue. We write $x := y$ for the function that maps x to y, and acts as the identity for the other variable in the target state. We omit guards true, trivial assignments, and parameters that not occur in the guard and are not touched by the assignment. Thus we write, for instance, Pop instead of Pop(p). Function $\mathcal{X}$ assigns variable set $\emptyset$ to locations l_0 and l_3, variable set $\{v\}$ to locations l_1, l_4 and l_6, and variable set $\{v, w\}$ to locations l_2, l_5 and l_7.

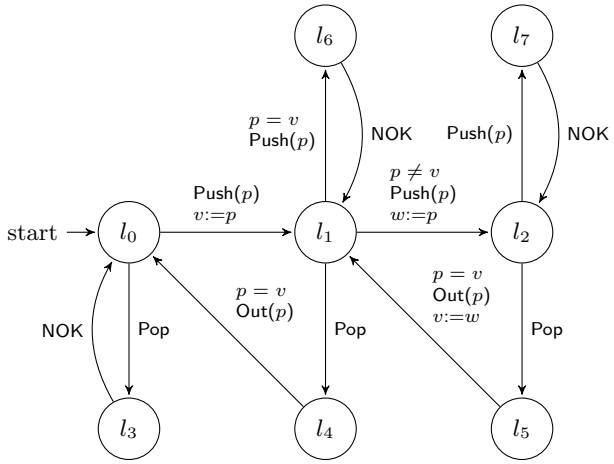


Fig. 8: FIFO-set with capacity 2.

Example 4. By just a minor change of the register automaton of Example 3, we may define a priority queue with capacity 2. This register automaton over the

structure $\langle \mathbb{R}, \{<\} \rangle$ is identical to the register automaton of Figure 8, except that the two outgoing Push-transitions of l_1 have been replaced by transitions

$$l_1 \xrightarrow{\mathsf{Push}, p<v, v:=p; w:=v} l_2 \qquad\qquad l_1 \xrightarrow{\mathsf{Push}, p\geq v, w:=p} l_2$$

This ensures that in location l_2, the value in register v is less than or equal to the value in register w. As a result, output Out will return the smallest value in the queue.

Semantically, a register automaton is just a finite representation of an infinite interface automaton.

Definition 12 (Semantics register automata). *Let $\mathcal{A} = \langle L, L_0, \mathcal{X}, \Sigma, \Gamma, \rightarrow \rangle$ be a register automaton. The interface automaton $\mathsf{RA2IA}(\mathcal{A})$ is the tuple $\langle Q, Q_0, \Sigma \times \mathcal{D}, \Gamma \times \mathcal{D}, \rightarrow' \rangle$, where*

1. *Q is the set of pairs $\langle l, \nu \rangle$ with $l \in L$ and $\nu : \mathcal{X}(l) \rightarrow \mathcal{D}$.*
2. *Q_0 is the set of pairs $\langle l, \nu \rangle \in Q$ with $l \in L_0$.*
3. *$\langle l, \nu \rangle \xrightarrow{a(d)}{}' \langle l', \nu' \rangle$ iff $\mathcal{A}$ has a transition $l \xrightarrow{a,g,\pi} l'$ such that g is satisfied in l and by parameter d (i.e., $\iota \models g$, where $\iota = \nu \cup \{(p, d)\}$), and $\nu' = \iota \circ \pi$.*

Two register automata $\mathcal{A}$ and $\mathcal{A}'$ are bisimilar *iff their associated interface automata are bisimilar, i.e., $\mathsf{RA2IA}(\mathcal{A}) \simeq \mathsf{RA2IA}(\mathcal{A}')$. Similarly, we call register automaton $\mathcal{A}$* deterministic *iff its associated interface automaton $\mathsf{RA2IA}(\mathcal{A})$ is deterministic.*

The interface automaton associated to the register automaton of Figure 8 is deterministic and, for instance, has the following sequence of transitions:

$$\langle l_0, \emptyset \rangle \xrightarrow{\mathsf{Push}(4)} \langle l_1, \{(v,4)\} \rangle \xrightarrow{\mathsf{Push}(5)} \langle l_2, \{(v,4),(w,5)\} \rangle \xrightarrow{\mathsf{Pop}} \langle l_5, \{(v,4),(w,5)\} \rangle$$
$$\xrightarrow{\mathsf{Out}(4)} \langle l_1, \{(v,5)\} \rangle \xrightarrow{\mathsf{Push}(5)} \langle l_6, \{(v,5)\} \rangle \xrightarrow{\mathsf{NOK}} \langle l_1, \{(v,5)\} \rangle.$$

Register automata over structure $\langle \mathbb{N}, \{=\} \rangle$ can be translated to a finite interface automaton by restricting the data domain $\mathcal{D}$ to a finite set. Let $\mathsf{RA2IA}_n(\mathcal{A})$ be the finite interface automaton obtained by replacing $\mathcal{D}$ by $\{0, \ldots, n-1\}$ in the definition of $\mathsf{RA2IA}(\mathcal{A})$, for any $n \in \mathbb{N}$. Heidarian [22, Chapter 8] showed that two register automata $\mathcal{A}$ and $\mathcal{A}'$ are bisimilar iff $\mathsf{RA2IA}_n(\mathcal{A}) \simeq \mathsf{RA2IA}_n(\mathcal{A}')$, for large enough n. Via the translations $\mathsf{RA2IA}_n$, each deterministic register automaton benchmark can be used to generate an infinite number of DFIA benchmarks, in which the numbers of states and transitions grow unboundedly. In several of our register automata benchmarks, inputs and outputs alternate. As a result, the DFIAs obtained via translations $\mathsf{RA2IA}_n$ are Mealy-like, and can subsequently be converted to Mealy machines via translation IA2Mealy from Section 2.5.

Thus far, all the register automaton benchmarks in our repository are deterministic register automata over structure $\langle \mathbb{N}, \{=\} \rangle$, but we are planning to include register automata benchmarks over different structures, such as the models described in [25].

3 Benchmarks Derived from Applications

Our repository `automata.cs.ru.nl` contains four types of benchmarks: (1) randomly generated automata, (2) small toy examples, (3) benchmarks derived from realistic applications, and (4) benchmarks obtained via the translations from Section 2. In this section, we focus on the benchmarks derived from realistic applications, and briefly pay attention to some of the smaller "toy" models that have been included in the repository. All the benchmarks in this section are either Mealy machines or register automata. In the next Section 4, we discuss algorithms for generating random automata, and a collection of randomly generated DFAs and Moore machines that we have included in the repository.

3.1 Mealy machines

The large majority of the Mealy machine benchmarks in our repository has fewer than 100 states, fewer than 20 inputs, and fewer than 50 outputs. For a detailled listing of the numbers of states, inputs and outputs of all the benchmarks we refer to `automata.cs.ru.nl/Table`.

Toy examples. We included several toy Mealy machines, such as a simple model of a coffee machine used as running example in [63], a trivial three state model used to explain L^* in [68], and some instructive examples from [44, 52].

Circuits. The logic synthesis workshops (LGSynth89, LGSynth91 and LGSynth93) provided 59 behavioral models for testing, logic synthesis and optimization of circuits, see [24, 12]. These models can be viewed as Mealy machines in several ways. We provide four interpretations of each model as a Mealy machine. If two or more interpretations give equivalent results, we have included only one of them in the repository. The circuit benchmarks have been used recently for Mealy machine testing by Hierons & Türker [31].

TCP. The Transmission Control Protocol (TCP) is a widely used transport layer protocol that provides reliable and ordered delivery of a byte stream from one computer application to another. The authors of [26] combined model learning and model checking in a case study involving Linux, Windows and FreeBSD implementations of TCP. Model learning was used to infer models of different software components and model checking was applied to fully explore what may happen when these components (e.g., a Linux client and a Windows server) interact. The analysis revealed several instances in which TCP implementations do not conform to their RFC specifications.

TLS protocol. TLS, short for Transport Layer Security, is a widely used protocol that aims to provide privacy and data integrity between two or more communicating computer applications, for example in HTTPS. The authors of [58] analyzed both server- and client-side implementations of TLS with a test harness that supports several key exchange algorithms and the option of client certificate

authentication. Using LearnLib, they succeeded to learn Mealy machine models of a number of TLS implementations. They showed that this approach can catch an interesting class of flaws that is apparently common in security protocol implementations: in three of the TLS implementations that were analyzed (GnuTLS, the Java Secure Socket Extension, and OpenSSL), new security flaws were found. This indicates that model learning is a useful technique to systematically analyze security protocol implementations. As the analysis of different TLS implementations resulted in different and unique state machines for each one, the technique can also be used for fingerprinting TLS implementations.

SSH protocol. SSH, short for Secure Shell, is a cryptographic network protocol that is widely used to interact securely with remote machines. The authors of [27] applied model learning to three SSH implementations (OpenSSH, Bitvise and DropBear) to infer Mealy machine models, and then used model checking to verify that these models satisfy basic security properties and conform to the RFCs. The analysis showed that all tested SSH server models satisfy the stated security properties. However, several violations of the standard were uncovered.

ABN AMRO e.dentifier2. The e.dentifier2 is a hand-held smart card reader with a small display, a numeric keyboard, and OK and Cancel buttons. Customers of the Dutch ABN AMRO bank use it for Internet banking in combination with a bank card and a PIN code. The authors of [16] showed that model learning can be successfully used to reverse engineer the behavior of the e.dentifier2, by using a Lego robot to operate the devices. The Mealy machines that were automatically inferred by the robot revealed a security vulnerability in the e.dentifier2, that was previously discovered by manual analysis, and confirmed the absence of this flaw in an updated version of this device.

EMV protocol. Bank cards (debit cards) are smart cards used for payment systems. Most smart cards issued by banks or credit cards companies adhere to the EMV (Europay-MasterCard-Visa) protocol standard, which is defined on top of ISO/IEC 7816. In [6], LearnLib and some simple abstraction techniques were used to learn Mealy machine models of EMV applications on bank cards issued by several Dutch banks (ABN AMRO, ING, Rabobank), one German bank (Volksbank), and one MasterCard credit cards issued by Dutch and Swedish banks (SEB, ABN AMRO, ING) and of one UK Visa Debit card (Barclays). These models provide a useful insight into decisions (or indeed mistakes) made in the design and implementation, and would be useful as part of security evaluations — not just for bank cards but for smart card applications in general — as they can show unexpected additional functionality that is easily missed in conformance tests.

MQTT protocol. The Message Queuing Telemetry Transport (MQTT) protocol is a lightweight publish/subscribe protocol that is well-suited for resource-constrained environments such as the Internet of Things (IoT). The authors of

[64] used model learning to obtain Mealy machine models of five freely available implementations of MQTT brokers (included in Apache ActiveMQ 5.13.3, emqttd 1.0.2, HBMQTT 0.7.1, Mosquitto 1.4.9 and VerneMQ 0.12.5p4). Examining these models, the authors found several violations of the MQTT specification. In fact, all but one of the considered implementations showed faulty behavior.

ESM printer controller. The Engine Status Manager (ESM) is a software component that is used in printers and copiers of Oce. Using a combination of LearnLib and a novel conformance testing algorithm, the authors of [62] succeeded to learn a Mealy machine model of this component fully automatically. Altogether, around 60 million queries were needed to learn a model of the ESM with 77 inputs and 3.410 states. They also constructed a model by flattening a Rational Rose Real-Time description from which the ESM software was generated, and established equivalence with the learned model.

An interventional X-ray system. Model learning and equivalence checking are used by [60] to improve a new implementation of a legacy control component. Model learning is applied to both the old and the new implementation of the Power Control Service (PCS) of an interventional X-ray system. The resulting models are compared using an equivalence check of a model checker. The authors report about their experiences with this approach at Philips. By gradually increasing the set of input stimuli, they obtained implementations of the PCS for which the learned behavior is equivalent.

From Rhapsody to Dezyne. In his PhD thesis, Schuts [59, Chapter 8] describes a case study, carried out at Philips, in which models created with a legacy tool (Rhapsody) are transformed to models that can be used by another tool (Dezyne). The transformation is established by means of a DSL for the legacy models. Model learning was applied to increase confidence in the correctness of the generated code. Two versions of state-machine code, generated by Rhapsody and Dezyne, were stimulated by all possible inputs and the resulting outputs were examined by LearnLib. The two models constructed by LearnLib were compared by the equivalence checker of the mCRL2 tool set. With this approach two errors were found in the Dezyne models that were not detected by the existing regression test set.

3.2 Register automata

Toy examples. We included several toy models in the repository: the sender and receiver of the well-known Alternating Bit Protocol, a simple login protocol, an automaton that test whether a list of numbers is a palindrome or a repdigit, and a river crossing puzzle.

SIP. The Session Initiation Protocol (SIP) is a signalling protocol used for initiating, maintaining, and terminating real-time sessions that include voice, video and messaging applications. In [4], an abstract Mealy machine model was inferred that describes the SIP Server entity when setting up connections with a SIP Client. The model was obtained by connecting LearnLib with the protocol simulator ns-2, and generated a model of the SIP component as implemented in ns-2. Using a (manually constructed) mapper component, concrete SIP messages were converted into abstract input and output symbols. Even though no implementation errors were found, the work of [4] showed the feasibility of the approach for inferring models of implementations of realistic communication protocols. In [2], the Mealy machine model of [4] was converted into a register automaton model that is included in the repository.

Data structures. As observed by Howar et al. [34], register automata with input and output events can be used to represent semantic interfaces of simple data structures such as stacks, queues, and FIFO-sets with fixed capacities. Since they are parametrized by their capacity, these data structures provide excellent benchmarks for model learning tools, see e.g., [3].

Biometric passport. The biometric passport is an electronic passport provided with a computer chip and antenna to authenticate the identity of travelers. Examples of used protocols are Basic Access Control (BAC), Active Authentication (AA) and Extended Access Control (EAC) [13]. Official standards are documented in the International Civil Aviation Organisation's (ICAO) Doc 9303 [36]. In [7], LearnLib was used to automatically generate a model of fragments of these protocols as implemented on an authentic biometric passport. The data on the chip could be accessed via a smart card reader with JMRTD serving as API. A simple mapper component serves as an intermediary between the SUT and LearnLib.

Bounded Retransmission Protocol. The Bounded Retransmission Protocol (BRP) is a well-known benchmark case study from the verification literature [30, 20]. The BRP is a variation of the classical alternating bit protocol that was developed by Philips to support infrared communication between a remote control and a television. In [5], a reference implementation of the protocol is described, as well as six faulty mutants of this implementation. The authors use a combination of model learning, model-based testing and verification to detect behavioral differences between the mutants and the reference implementation.

4 Random Generation of Benchmarks

As argued throughout this paper, high-quality benchmarks are an integral part of the evaluation of (automata learning) algorithms. In this context, synthetic, i.e., randomly generated, automata play an important role due to their relevancy to average case analyses and their usually high Kolmogorov complexity [45, 17].

Contrary to what one might think, however, randomly generating automata is not a trivial task: automata carry a semantics (in form of the accepted language) and, hence, properties such as connectedness and minimality with respect to the accepted language are of great importance. In fact, estimating the number of pairwise non-equivalent automata of a certain size is already a challenging problem [29, 17].

In this section, we survey three popular algorithms for generating random DFAs, taken both from the literature and from automata learning competitions:

1. the algorithm used in the Abbadingo DFA learning competition [42], which we present in Section 4.1;
2. the algorithm used in the Stamina DFA learning competition [70, 71] (based on the forest-fire algorithm [45] for generating random graphs), which we present in Section 4.2; and
3. Champarnaud and Paranthoën's method [17], which we present in Section 4.3.

Methods for generating other types of state machines (such as NFAs, Mealy and Moore machines, etc.) exists as well, but are often ad-hoc approaches and far less studied.

In Section 4.4, we briefly describe a series of random DFAs and random Moore machines, which we have generated on the occasion of Bernhard Steffen's 60^{th} birthday. In this section, we also sketch a simple method for randomly generating Moore machines.

For the following description, recall from Section 2.2 that a DFA is a tuple $\mathcal{A} = \langle Q, Q_0, \Sigma, \to, F \rangle$ where $\langle Q, Q_0, \Sigma, \to \rangle$ is a complete and deterministic FSM and $F \subseteq Q$ is a set of final states. Moreover, let $n = |Q|$ denote the desired size, i.e., the number of states, of the DFA to be generated.

4.1 Abbadingo Competition Random DFA Algorithm

The Abbadingo random DFA algorithm [42] is a simplistic algorithm, which constructs a DFA with n states ($n > 0$) in four steps:

1. It creates n states, say $Q = \{q_1, \ldots, q_n\}$.
2. For each pair of state $p \in Q$ and input symbol $a \in \Sigma$, it chooses a destination state $q \in Q$ uniformly at random and adds the transition $p \xrightarrow{a} q$.
3. It chooses a state $q_0 \in Q$ uniformly at random and marks it as the initial state, i.e., $Q_0 = \{q_0\}$.
4. For each state $q \in Q$, it determines whether q is a final state by flipping a fair coin, i.e., it adds q to F with probability $1/2$.

Clearly, a major drawback of this simple approach is the neglect of any structural property of the generated DFA—except for the fact that the resulting automaton is deterministic. In particular, the algorithm neither guarantees that the resulting DFA is *accessible*, i.e., that all its states are reachable from the initial state, nor that it is minimal. For this reason, the Abbadingo competition

used the following procedure: in order to obtain a DFA of size roughly n, a DFA of size $1.2n$ is generated and all states that are not reachable from the initial state are removed. Although this additional step ensures that the resulting DFA is accessible, it might still not produce minimal DFAs.

4.2 Stamina Competition Random DFA Algorithm

The algorithm used in the Stamina competition [70] has been designed to produce random DFAs that are representative of software models. Its basis is the forest-fire algorithm by Leskovec, Kleinberg, and Faloutsos [45], which produces directed graphs that resemble complex networks arising in a variety of domains. The forest-fire algorithm is an iterative algorithm (each iteration adds one new vertex as well as edges from and to this vertex) that takes three parameters as input: a number $N > 0$ of vertices, a *forward burning probability* $p \in [0, 1]$, and a *backward burning ratio* $r \in [0, 1]$.

The forest-fire algorithm proceeds in N rounds. In the first round, it initializes the graph with a single vertex. In each subsequent round, it performs the following four steps (Step 1 inserts a new vertex, while Steps 2, 3, and 4 insert new edges):

1. The algorithm creates a new vertex v. Moreover, it initializes an auxiliary set $U = \emptyset$, which is used to mark vertices that have been visited by the algorithm in the current round.
2. It picks a vertex $w \neq v$, called *ambassador vertex*, uniformly at random and adds the edge $v \to w$. Moreover, it adds w to U, marking w as visited.
3. It draws a random number $x \in \mathbb{N}$ from a geometric distribution with mean $p/(1-p)$ and a second random number $y \in \mathbb{N}$ from a geometric distribution with mean $rp/(1-rp)$. Then, it selects
   - x incoming edges of w, say $v_1 \to w, \ldots, v_x \to w$, and
   - y outgoing edges of w, say $w \to v'_1, \ldots, w \to v'_y$,

   uniformly at random such that $\{v_1, \ldots, v_x, v'_1, \ldots, v'_y\} \cap U = \emptyset$, i.e., none of the vertices $v_1, \ldots, v_x$ and $v'_1, \ldots, v'_y$ have been visited in this iteration; if not enough edges are available, the algorithm selects as many as possible.
4. It adds the edges $v \to v_1, \ldots, v \to v_x, v \to v'_1, \ldots, v \to v'_x$ and then applies Step 2 recursively with each of the vertices $v_1, \ldots, v_x, v'_1, \ldots, v'_y$ as ambassador vertex. Note that this procedure stops eventually as vertices cannot be visited more than once.

To generate a DFA (rather than a directed graph), the algorithm used in the Stamina competition takes three additional parameters as input: a set Σ of input symbols, a *self-loop probability* $l \in [0, 1]$, and a *parallel-edge probability* $e \in [0, 1]$. (Note that the forest-fire algorithm can neither create self-loops nor parallel edges.) Based on these additional parameters, the forest-fire algorithm is adapted as follows:

- The initial state is chosen uniformly at random, and each vertex has the probability $1/2$ of being a final state.

– In order to make sure that each state is reachable, edges added in Step 2 are added in the reverse direction, i.e., $w \to v$.
– Whenever the forest-fire algorithm adds an edge in Step 4, with probability l this edge gets instead redirected to form a self-loop.
– Whenever the forest-fire algorithm inserts an edge, the edge is turned into a transition that is labeled with an input symbol $a \in \Sigma$. The input symbol a is drawn uniformly at random from the set Σ such that the automaton remains deterministic, i.e., symbols that are already used in an outgoing transition from the state in question are not considered. If all symbols from Σ already occur on an outgoing transition, then no transition is added.
– Finally, every time a transition is inserted, a second, parallel transition is added with probability e. The second transition is labeled using the labeling rule described above.

Although this algorithm produces accessible DFAs, it does not guarantee that these DFAs are minimal. To account for this, the DFAs used in the Stamina competition have been generated slightly larger than desired and have subsequently been minimized. The parameters used to generate the competition DFAs were $\Sigma = \{1, \ldots, a\}$ for $a \in \{2, 5, 10, 20, 50\}$, $n = 50$ (the actual value N has been chosen slightly larger than 50 due to the subsequent minimization process), $f = 0.31$, $r = 0.385$, $l = 0.2$, and $e = 0.2$.

4.3 Champarnaud and Paranthoën's Method

Champarnaud and Paranthoën's method [17] is a generalization of an algorithm proposed by Nicaud [53], which randomly generates accessible DFAs over two input symbols. An interesting property of Nicaud's algorithm is that it generates minimal DFAs with a probability of about $4/5$. Champarnaud and Paranthoën's method shares this property when generating DFAs over two input symbols, while an experimental evaluation with over a million DFAs has shown that nearly all generated DFAs were minimal if the number of input symbols was chosen greater than two [17]. Hence, should a minimal DFA be required, a viable approach is to simply repeat Champarnaud and Paranthoën's method until the resulting DFA is minimal.

Champarnaud and Paranthoën's method is a fairly complex algorithm, which is based on two ideas:

– The FSM $\langle Q, Q_0, \Sigma, \to \rangle$ underlying any DFA can be represented by a Σ-labeled tree of arity $m = |\Sigma|$ with $n = |Q|$ inner nodes, i.e., a tree of arity m with n inner nodes whose edges are labeled with symbols from Σ.
– Labeled trees can be encoded by a special type of tuples over the natural numbers, which Champarnaud and Paranthoën call *generalized tuples*.

Hence, one can generate a random DFA by first randomly generating a generalized tuple, then constructing the corresponding tree, and finally deriving a DFA from the tree. Although an in-depth description of this procedure is out of the

scope of this paper, the remainder of this section sketches the main steps of the algorithm.

At the heart of Champarnaud and Paranthoën's method lies the observation that every Σ-labeled tree is determined (up to isomorphism) by one of its *prefix traversals*. More precisely, a complete m-ary tree with n inner nodes—and, therefore, $s = n(m-1)+1$ leaf nodes—can be encoded by the tuple

$$(k_1, \ldots, k_{s-1}) \in \{1, \ldots, n\}^{s-1},$$

where the i-th entry k_i corresponds to the number of inner nodes visited during a prefix traversal of the tree prior to the visit of the i-th leaf (note that there is no need to store this information for the last leaf as this number is n). The set of all generalized tuples of length $l+1$ can be constructed recursively from the set of generalized tuples of length l, and Champarnaud and Paranthoën give an algorithm to draw such tuples randomly. Once a generalized tuple of length $s-1$ has been generated, the corresponding tree with n inner nodes can be constructed effectively.

The tree generated in the previous step represents a deterministic transition structure that serves as template for a number of (non-isomorphic) n-state DFAs. Constructing a DFA from such a template involves two steps: first, edges to leaf nodes need to be redirected to inner nodes (so as to be able to produce a DFA that is complete and accepting an infinite language); second, final states have to be selected. However, edges cannot be redirected arbitrarily as this might result in the same DFA being generated from two different generalized tuples. In order to prevent this from happening, Champarnaud and Paranthoën's method redirects edges only to inner nodes that have been visited earlier during the prefix traversal. The final DFA is then obtained by setting the initial state to be the root node and choosing uniformly at random one possibility of inserting back edges and selecting final states. Note that this implies in particular that the probability of a state being final is $1/2$.

4.4 Random DFAs and Moore Machines Dedicated to Bernhard Steffen's 60th Birthday

On the occasion of Bernhard Steffen's 60th birthday, we have included four sets of randomly generated DFAs and Moore machines in our repository:

1. 60×60 DFAs with 1 000 states each over the alphabet $\Sigma = \{0, 1, \ldots, 19\}$;
2. 60×60 DFAs with 2 000 states each over the alphabet $\Sigma = \{0, 1, \ldots, 9\}$;
3. 60×60 Moore machines with 1 000 states each over the input alphabet $\Sigma = \{0, 1, \ldots, 19\}$ and output alphabet $\Gamma = \Sigma$; and
4. 60×60 Moore machines with 2 000 states each over the input alphabet $\Sigma = \{0, 1, \ldots, 9\}$ and output alphabet $\Gamma = \Sigma$.

The number of states and the number of elements in the input/output alphabets of these automata were chosen to be challenging, though still manageable for state-of-the-art algorithms.

All DFAs were generated using libalf's [11] off-the-shelf implementation of Champarnaud and Paranthoën's method. To generate Moore machines, we used the following two-step process: first, we randomly generated a DFA using Champarnaud and Paranthoën's method, which serves as the LTS underlying our Moore machines; second, we assigned to each state an output symbol that was drawn uniformly at random from the output alphabet. Note that the second step is in fact a generalization of the way Champarnaud and Paranthoën select final states, which is essentially by flipping a fair coin for each state. As with all methods described in this section, however, our DFAs and Moore machines are accessible but might not be minimal.

5 Conclusions

Many of the benchmark models in our repository have clear practical relevance, e.g., they helped to reveal standard violations in network protocols and eliminate bugs in industrial software. Nevertheless, the benchmarks are surprisingly small: several models have less than ten states and our largest models only have a few thousand states. A possible explanation is that model learning and testing typically focus on a single component (e.g., a TCP server) and there is already some implicit abstraction in the selection of the interface. This should be contrasted with benchmarks used for explicit model checking, which typically focus on the behavior of networks of components, and have millions of states.

Even though our benchmarks are small, they still pose enormous challenges for state-of-the-art automata learning and conformance testing tools. In practice, conformance testing algorithms often have difficulties to find subtle bugs in implementations for models with more than say a hundred states and a dozen inputs. For instance, with 3.410 states and 77 inputs the ESM printer controller model is at the limit of what current algorithms can handle [62]. In particular, state-of-the-art techniques are unable to learn models of the printer controller for slighty different configurations of the same software. Also, input/output interactions and resets of software and hardware often take a significant amount of time. For instance, in the case study of the interventional X-ray system [60], it took up to 9 hours to learn models with up to 9 states and 12 inputs. This was because running a single test sequence took on average about 10 seconds and a reset of the implementation took about 5 seconds. This means that any reduction of the number of queries needed for learning and testing reliable models has immediate practical relevance. Clearly, a comprehensive evaluation of existing learning and testing algorithms on our benchmarks is an important direction for future research.

Finally, we would like to encourage all our colleagues to contribute new benchmarks to the repository! Our automata wiki is built using the PmWiki software, which makes it easy to add new benchmarks.

Acknowledgements. This article was initiated at the Dagstuhl Seminar 16172 "Machine Learning for Dynamic Software Analysis: Potentials and Limits" organized by Amel Bennaceur, Reiner Hähnle, and Karl Meinke. We thank Fides

Aarts, Petra van den Bos, Alexander Fedotov, Paul Fiterău-Broştean, Falk Howar, Joshua Moerman, Erik Poll, and Joeri de Ruiter for helping with the repository. Many thanks to Pierre van de Laar and the anonymous reviewers for their suggestions on an earlier version of this paper.

References

1. F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F.W. Vaandrager. Learning register automata with fresh value generation. In *Proceedings ICTAC*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015.
2. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *Proceedings FM 2012*, volume 7436 of *LNCS*, pages 10–27. Springer, August 2012.
3. F. Aarts, F. Howar, H. Kuppens, and F.W. Vaandrager. Algorithms for inferring register automata - A comparison of existing approaches. In *Proceedings ISoLA*, volume 8802 of *LNCS*, pages 202–219. Springer, 2014.
4. F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proceedings ICTSS*, volume 6435 of *LNCS*, pages 188–204. Springer, 2010.
5. F. Aarts, H. Kuppens, G.J. Tretmans, F.W. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1–2):189–224, 2014.
6. F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop, IEEE International Conference on*, pages 461–468, Los Alamitos, CA, USA, 2013. IEEE Computer Society.
7. F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In *Proceedings ISoLA 2010*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.
8. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
9. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *Proceedings FASE 2005*, volume 3442 of *LNCS*, pages 175–189. Springer, 2005.
10. B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In Craig Boutilier, editor, *Proceedings IJCAI 2009*, pages 1004–1009, 2009.
11. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D.R. Piegdon. libalf: The automata learning framework. In *Proceedings CAV*, volume 6174 of *LNCS*, pages 360–364. Springer, 2010.
12. F. Brglez. ACM/SIGDA benchmark dataset, 1996. Available through URL `http://people.engr.ncsu.edu/brglez/CBL/benchmarks/Benchmarks-upto-1996.html`. Last accessed on August 14, 2018.
13. BSI. Advanced security mechanisms for machine readable travel documents - extended access control (eac) - version 1.11. Technical Report TR-03110, German Federal Office for Information Security (BSI), Bonn, Germany, 2008.
14. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.*, 84(1):54–66, 2015.
15. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

16. G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings WOOT'14*, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.
17. J.-M. Champarnaud and T. Paranthoën. Random generation of DFAs. *Theor. Comput. Sci.*, 330(2):221–235, 2005.
18. D. van Dalen. *Logic and Structure*. Springer-Verlag, 1983.
19. L. D'Antoni. AutomatArk. Available through URL `https://github.com/lorisdanto/automatark`, last accessed August 14, 2018.
20. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proceedings TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer-Verlag, April 1997.
21. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings ESEC/FSE-01*, volume 26 of *Software Engineering Notes*, pages 109–120, New York, September 2001. ACM Press.
22. F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. PhD thesis, Radboud University Nijmegen, July 2012.
23. R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, and N. Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, 52(12):1286–1297, 2010.
24. P. Fiser. Collection of digital design benchmarks. Available through URL `https://ddd.fit.cvut.cz/prj/Benchmarks/`. Last accessed on August 14, 2018.
25. P. Fiterău-Broştean and F. Howar. Learning-based testing the sliding window behavior of TCP implementations. In *Proceedings FMICS-AVoCS 2017*, volume 10471 of *LNCS*, pages 185–200. Springer, 2017.
26. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *Proceedings CAV'16*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.
27. P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *Proceedings SPIN Symposium*, SPIN 2017, pages 142–151, New York, NY, USA, 2017. ACM.
28. E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Softw., Pract. Exper.*, 30(11):1203–1233, 2000.
29. F. Harary and E.M. Palmer. Enumeration of finite automata. *Information and Control*, 10(5):499–508, 1967.
30. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In *Proceedings TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer-Verlag, 1994.
31. R.M. Hierons and U.C. Türker. Incomplete distinguishing sequences for finite state machines. *Comput. J.*, 58(11):3089–3113, 2015.
32. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
33. F. Howar. *Active learning of interface programs*. PhD thesis, University of Dortmund, June 2012.
34. F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *Proceedings ISoLA*, volume 7609 of *LNCS*, pages 554–571. Springer, 2012.
35. F. Howar and B. Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 123–148. Springer, 2018.

36. ICAO. Doc 9303 - machine readable travel documents - part 1-2. Technical report, International Civil Aviation Organization, 2006. Sixth edition.
37. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Proceedings RV 2014*, pages 307–322, Cham, 2014. Springer.
38. M. Jasper, M. Fecke, B. Steffen, M. Schordan, J. Meijer, J. van de Pol, F. Howar, and S.F. Siegel. The RERS 2017 challenge and workshop (invited paper). In *Proceedings SPIN Symposium*, pages 11–20. ACM, 2017.
39. B. Jonsson. Modular verification of asynchronous networks. In PODC'87 [54], pages 152–166.
40. M.J. Kearns and U.V. Vazirani. *An introduction to computational learning theory*. MIT Press, 1994.
41. R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.
42. K.J. Lang, B.A. Pearlmutter, and R.A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings ICGI-98*, volume 1433 of *LNCS*, pages 1–12. Springer, 1998.
43. D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.
44. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
45. J. Leskovec, J.M. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *Transactions on Knowledge Discovery from Data*, 1(1), 2007.
46. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.
47. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In PODC'87 [54], pages 137–151. A full version is available as MIT Technical Report MIT/LCS/TR-387.
48. G.H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955.
49. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *Proceedings TACAS*, volume 6605 of *LNCS*, pages 220–223. Springer, 2011.
50. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings POPL 2017*, pages 613–625. ACM, 2017.
51. E.F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.
52. K. Naik. Efficient computation of unique input/output sequences in finite-state machines. *IEEE/ACM Trans. Netw.*, 5(4):585–599, August 1997.
53. C. Nicaud. *Étude du comportement en moyenne des automates finis et des langages rationnels*. PhD thesis, Université Paris 7, 2000.
54. *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, August 1987.
55. H. Raffelt, B. Steffen, and T. Berg. LearnLib: a library for automata learning and experimentation. In *Proceedings FMICS'05*, pages 62–71, New York, NY, USA, 2005. ACM Press.
56. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
57. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

58. J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *Proceedings USENIX Security 15*, pages 193–206. USENIX Association, August 2015.
59. M. Schuts. *Industrial Experiences in Applying Domain Specific Languages for System Evolution*. PhD thesis, Radboud University Nijmegen, September 2017.
60. M. Schuts, J. Hooman, and F.W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *Proceedings iFM*, volume 9681 of *Lecture Notes in Computer Science*, pages 311–325, 2016.
61. M. Shahbaz and R. Groz. Inferring Mealy machines. In *Proceedings FM 2009*, volume 5850 of *LNCS*, pages 207–222. Springer, 2009.
62. W. Smeenk, J. Moerman, F.W. Vaandrager, and D.N. Jansen. Applying automata learning to embedded control software. In *Proceedings ICFEM 2015*, volume 9407 of *LNCS*, pages 67–83. Springer, 2015.
63. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Proceedings SFM 2011*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.
64. M. Tappler, B.K. Aichernig, and R. Bloem. Model-based testing IoT communication via active automata learning. In *Proceedings ICST 2017*, pages 276–287. IEEE Computer Society, 2017.
65. J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17:103–120, 1996.
66. J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
67. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
68. F.W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, February 2017.
69. M. Volpato and J. Tretmans. Towards quality of model-based testing in the ioco framework. In *Proceedings JAMAICA 2013*, pages 41–46, New York, NY, USA, 2013. ACM.
70. N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont. A framework for the competitive evaluation of model inference techniques. In *Proceedings MIIT '10*, pages 1–9. ACM, 2010.
71. N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont. STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empirical Software Engineering*, 18(4):791–824, 2013.