DOI:10.1145/2967606

Model learning emerges as an effective method for black-box state machine models of hardware and software components.

BY FRITS VAANDRAGER

Model Learning

WE ROUTINELY MANAGE to learn the behavior of a device or computer program by just pressing buttons and observing the resulting behavior. Especially children are very good at this and know exactly how to use a smartphone or microwave oven without ever consulting a manual. In such situations, we construct a mental model or state diagram of the device: through experiments we determine in which global states the device can be and which state transitions and outputs occur in response to which inputs. This article is about the design and application of algorithms that perform this task automatically.

There are numerous approaches where models of software components are inferred through analysis of the code, mining of system logs, or by performing tests. Many different types of models are inferred, for example, hidden Markov models, relations between variables, and class diagrams. In this article, we focus on one specific type of models, namely *state diagrams*, which are crucial for understanding the behavior of many software systems, such as (security and network) protocols and embedded control software. Model inference techniques can be either white box or black box, depending on whether they need access to the code. In this article, we discuss *black box* techniques. Advantages of these techniques are that they are relatively easy to use and can also be applied in situations where we do not have access to the code or to adequate white box tools. As a final restriction, we only consider techniques for *active learning*, that is, techniques that accomplish their task by actively doing experiments (tests) on the software. There is also an extensive body of work on passive learning, where models are constructed from (sets of) runs of the software. An advantage of active learning is that it provides models of the full behavior of a software component, and not just of the specific runs that have occurred during actual operation.

The fundamental problem of active, black-box learning of state diagrams (or automata) has been studied for decades. In 1956, Moore[31] first

» key insights

- **Model learning aims to construct black-box state diagram models of software and hardware systems by providing inputs and observing outputs. The design of algorithms for model learning constitutes a fundamental research problem.**
- **Recently, much progress has been made in the design of new algorithms, both in a setting of finite state diagrams (Mealy machines) and in richer settings with data (register automata). Through the use of abstraction techniques, these algorithms can be applied to complex systems.**
- **Model learning is emerging as a highly effective bug-finding technique, with applications in areas such as banking cards, network protocols, and legacy software.**

ILLUSTRATION BY MARIE DOMMENGET

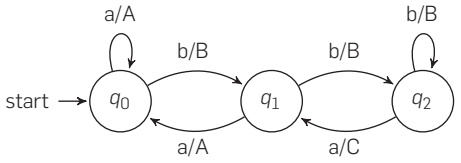

l1
l0
MODEL
CHECKING

Mealy Machines

A *(deterministic) Mealy machine* is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where I is a finite set of inputs, O is a finite set of outputs, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times I \to Q$ is a transition function, and $\lambda : Q \times I \to O$ is an output function.

Figure 1 gives a graphical representation of a simple Mealy machine with inputs $\{a, b\}$, outputs $\{A, B, C\}$, states $\{q_0, q_1, q_2\}$, and initial state q_0.

Figure 1. A simple Mealy machine.



Output function λ is extended to sequences of inputs by defining, for all $q \in Q$, $i \in I$, and $\sigma \in I^*$, $\lambda(q, \epsilon) = \epsilon$, and $\lambda(q, i\sigma) = \lambda(q, i)\lambda(\delta(q, i), \sigma)$. The behavior of Mealy machine $\mathcal{M}$ is defined by function $A_{\mathcal{M}} : I^* \to O^*$ with $A_{\mathcal{M}}(\sigma) = \lambda(q^0, \sigma)$, for $\sigma \in I^*$. Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are *equivalent*, denoted $\mathcal{M} \approx \mathcal{N}$, iff $A_{\mathcal{M}} = A_{\mathcal{N}}$. Sequence $\sigma \in I^*$ *distinguishes* $\mathcal{M}$ and $\mathcal{N}$ if and only if $A_{\mathcal{M}}(\sigma) \neq A_{\mathcal{N}}(\sigma)$.

proposed the problem of learning finite automata, provided an exponential algorithm and proved that the problem is inherently exponential. The problem has been studied under different names by different communities: control theorists refer to it as system identification, computation linguists speak about grammatical inference,[22] some papers use the term regular inference,[8] regular extrapolation,[20] or active automata learning,[24] and security researchers coined the term protocol state fuzzing.[34] Here, we will use the term *model learning* in analogy with the commonly used term model checking.[15] Whereas model checking is widely used for analyzing finite-state models, model learning is a complementary technique for building such models from observed input–output data.

In 1987, Angluin[6] published a seminal paper in which she showed that finite automata can be learned using the so-called *membership* and *equivalence queries*. Even though faster algorithms have been proposed since then, the most efficient learning algorithms that are being used today all follow Angluin's approach of a *minimally adequate teacher (MAT)*. In the MAT framework, learning is viewed as a game in which a learner has to infer the behavior of an unknown state diagram by asking queries to a teacher. The teacher knows the state diagram, which in our setting is a Mealy machine $\mathcal{M}$ (see Mealy machines for the definition). Initially, the learner only knows the inputs I and outputs O of $\mathcal{M}$. The task of the learner is to learn $\mathcal{M}$ through two types of queries:

- With a *membership query* (MQ), the learner asks what the output is in response to an input sequence $\sigma \in I^*$. The teacher answers with output sequence $A_{\mathcal{M}}(\sigma)$.
- With an *equivalence query* (EQ), the learner asks if a hypothesized Mealy machine $\mathcal{H}$ with inputs I and outputs O is correct, that is, whether $\mathcal{H}$ and $\mathcal{M}$ are equivalent. The teacher answers *yes* if this is the case. Otherwise she answers *no* and supplies a *counterexample* $\sigma \in I^*$ that distinguishes $\mathcal{H}$ and $\mathcal{M}$.

The L^* algorithm of Angluin[6] is able to learn Mealy machine $\mathcal{M}$ by asking a polynomial number of membership and equivalence queries (polynomial in the size of the corresponding canonical Mealy machine). In the Angluin's algorithm, we give a simplified presentation of the L^* algorithm. Actual implementations, for instance in LearnLib[26] and libalf,[9] contain many optimizations.

Peled et al.[19,32] made the important observation that the MAT framework can be used to learn black box models of software and hardware components. Suppose we have a component, which we call the *System Under Learning (SUL)*, whose behavior can be described by (an unknown) Mealy machine $\mathcal{M}$. Suppose further that it is always possible to bring the SUL back to its initial state. A membership query can now be implemented by bringing the SUL to its initial state and then observing the outputs generated by the SUL in response to the given input sequence. Equivalence query can be approximated using a conformance testing (CT) tool[29] via a finite number of *test queries* (TQs). A test query asks for the response of the SUL to an input sequence, similar to a membership query. If one of the test queries exhibits a counterexample then the answer to the equivalence query is *no*, otherwise the answer is *yes*. A schematic overview is shown in Figure 4. In this approach, the task of the learner is to construct hypotheses, whereas the task of the conformance testing tool is to test the validity of these hypotheses. As a testing tool can only pose a finite number of queries, we can never be sure that a learned model is correct. However, a finite and complete conformance test suite does exists if we assume a bound on the number of states of machine $\mathcal{M}$.[29]

The pioneering work of Peled et al.[32] and Steffen et al.[8,20,23] established fascinating connections between model learning and the area of formal methods, in particular model checking and model-based testing. Subsequent research has confirmed that, in the absence of a tractable white box model of a reactive system, a learned model is often an excellent alternative that may be obtained at relatively low cost.

In order to check properties of learned models, model checking[15] can be used. In fact, Peled et al.[32] showed how model learning and model checking can be fully integrated in an approach called *black box checking*. The basic idea is to use a model checker as a "preprocessor" for the conformance testing tool in Figure 4. When the teacher receives a hypothesis from the learner, it first runs a model checker to verify if the hypothesis model satisfies all the properties from the SUL's specification. Only if this is true the

hypothesis is forwarded to the conformance tester. If one of the properties does not hold then the model checker produces a counterexample. Now there are two cases. The first possibility is that the counterexample can be reproduced on the SUL. This means we have demonstrated a bug in the SUL (or in its specification) and we stop learning. The second possibility is that the counterexample cannot be reproduced on the SUL. In this case the teacher returns the counterexample to the learner since it follows that the hypothesis is incorrect. In later work,[16,19] the black box checking approach has been further refined and it has been successfully applied to several industrial cases.

The required number of membership queries of most learning algorithms grows linearly with the number of inputs and quadratically with the number of states.[24] This means that learning algorithms scale rather well when the number of inputs grows; in other words, formulating a new hypothesis is easy. However, checking that a hypothesis is correct (conformance testing), quickly becomes a bottleneck for larger numbers of inputs. If the current hypothesis has n states, the SUL has n' states, and there are k inputs, then in the worst case we need to run test sequences that contain all possible sequences of $n' - n$ inputs, that is, $k^{(n'-n)}$ possibilities.[29] As a result, model learning currently can only be applied if there are less than, say, 100 inputs. Thus, we seek methods that help us to reduce the number of inputs.

Abstraction is the key for scaling model learning methods to realistic applications. Cho et al.[14] succeeded to infer models of realistic botnet command and control protocols by placing an emulator/mapper between botnet servers and the learning software, which concretizes the alphabet symbols into valid network messages and sends them to botnet servers. When responses are received, the emulator does the opposite—it abstracts the response messages into the output alphabet and passes them on to the learning software. A schematic overview of this learning setup is shown in Figure 5. The idea of an intermediate mapper component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning. Aarts et al.[2] developed a mathematical theory of such intermediate abstractions, with links to predicate abstraction and abstract interpretation.

A complementary, simple but practical approach is to apply model learning for multiple smaller subsets of inputs. This will significantly reduce the learning complexity, also because the set of reachable states will typically

Angluin's Algorithm

The L^* algorithm incrementally constructs an observation table with entries taken from the set O of outputs. The rows are labeled by words in $S \cup S \cdot I$, where S is a nonempty finite prefix-closed language, and the columns by a nonempty finite suffix-closed language E. Formally, an observation table is a triple (S, E, row), where row: $S \cup (S \cdot I) \rightarrow (E \rightarrow O)$. For a given prefix w and suffix e, $row(w)(e)$ returns the last output produced by the SUL in response to the membership query we. Initially, S only contains the empty word ϵ, and E equals set of inputs I.

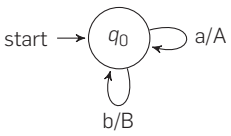
Two crucial properties of the observation table allow for the construction of a Mealy machine: closedness and consistency. Observation table (S, E, row) is *closed* if for all $w \in S \cdot I$ there is a $w' \in S$ with $row(w) = row(w')$. It is *consistent* if whenever $row(w_1) = row(w_2)$ for some $w_1, w_2 \in S$, then $row(w_1 a) = row(w_2 a)$ for all $a \in I$.

If a table is closed and consistent, the learner constructs a Mealy machine $\mathcal{H} = (I, O, Q, q_0, \delta, \lambda)$ with $Q = \{row(w) \mid w \in S\}$, $q_0 = row(\epsilon)$, $\delta(row(w), a) = row(w \cdot a)$, and $\lambda(row(w), a) = row(w)(a)$.

Assume the teacher knows the Mealy machine $\mathcal{M}$ from Figure 1. The learner starts to ask queries to fill the initial table. The result is shown in Figure 2 (left). As this table is both closed and consistent, the learner constructs an initial hypothesis $\mathcal{H}$, shown in Figure 2 (right).

Figure 2. First table and hypothesis $\mathcal{H}$.

$\mathcal{O}_1$	a	b
ϵ	A	B
a	A	B
b	A	B



Hypothesis $\mathcal{H}$ is incorrect since, for instance, sequence bba distinguishes $\mathcal{H}$ from $\mathcal{M}$. Assume that the teacher returns counterexample bba to the learner. To process this counterexample, the learner adds bba and all its prefixes to S and constructs the table shown in Figure 3 (left). Since $row(\epsilon) = row(b)$ but $row(b)(a) \neq row(bb)(a)$ this table is not consistent. Thus, we add ba to set E and obtain the table shown in Figure 3 (right). This table is closed and consistent, and the corresponding Mealy machine is equivalent to $\mathcal{M}$.

Figure 3. Second and third table.

$\mathcal{O}_2$	a	b
ϵ	A	B
b	A	B
bb	C	B
bba	A	B
a	A	B
ba	A	B
bbb	C	B
bbaa	A	B
bbab	C	B

$\mathcal{O}_3$	a	b	ba
ϵ	A	B	A
b	A	B	C
bb	C	B	C
bba	A	B	C
a	A	B	A
ba	A	B	A
bbb	C	B	C
bbaa	A	B	A
bbab	C	B	C

be smaller for a restricted number of stimuli. Models learned for a subset of the inputs may then be used to generate counterexamples while learning models for larger subsets. Yet another approach, which, for instance, has been applied by Chalupar et al.,[13] is to merge several input actions that usually occur in a specific order into a single high-level action, thus reducing the number of inputs. Again, models that have been learned with a small number of high level inputs may be used to generate counterexamples in subsequent experiments in which these inputs are broken up into their constituents.

Paraphrasing C.A.R. Hoare, one could say that in every large program there is a small state machine trying to get out. By choosing a proper set of input actions and by defining an appropriate mapper/abstraction, we can make this small state machine visible to the learner.

Figure 4. Model learning within the MAT framework.

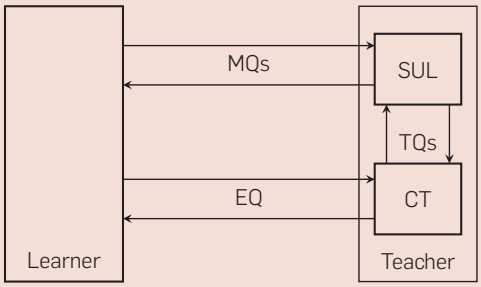


Figure 5. Model learning with a mapper.

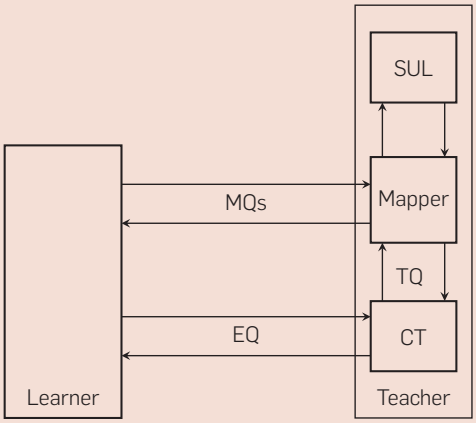
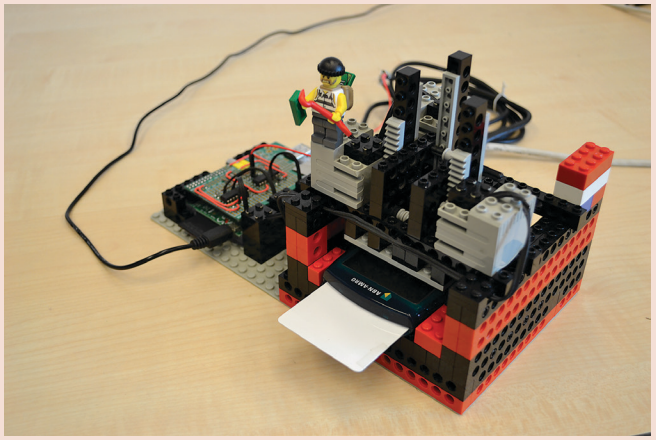


Figure 6. Lego robot used to reverse engineer the e.dentifier2 smartcard reader (picture courtesy of Chalupar[13]).



Examples of Applications

During recent years, model learning has been successfully applied to numerous practical cases in different domains. There have been industrial applications, for instance, on regression testing of telecommunication systems at Siemens,[20] on integration testing at France Telecom,[36] on automatic testing of an online conference service of Springer Verlag,[39] and on testing requirements of a brake-by-wire system from Volvo Technology.[16] Below, I review some representative case studies that have been carried out at Radboud University related to smart cards, network protocols, and legacy software.

Smartcards. Chalupar et al.[13] used model learning to reverse engineer the e.dentifier2, a smartcard reader for Internet banking. To be able to learn a model of the e.dentifier2, the authors constructed a Lego robot, controlled by a Raspberry Pi that can operate the keyboard of the reader (see Figure 6). Controlling all this from a laptop, they then could use LearnLib[26] to learn models of the e.dentifier2. They learned a four-state Mealy machine of one version of the e.dentifier2 that revealed the presence of a security flaw, and showed that the flaw is no longer present in a three-state model for the new version of the device.

In another study, Aarts et al.[3] learned models of implementations of the EMV protocol suite on bank cards issued by several Dutch and German banks, on MasterCard credit cards issued by Dutch and Swedish banks, and on one UK Visa debit card. To learn the models, LearnLib performed between 855 and 1,696 membership and test queries for each card and produced models with four to eight states. (Figure 7 shows one of the learned models.) All cards resulted in different models, only the applications on the Dutch cards were identical. The models learned did not reveal any security issues, although some peculiarities were noted. The authors argue that model learning would be useful as part of security evaluations.

Network protocols. Our society has become completely dependent on the

correct functioning of network and security protocols. Bugs or vulnerabilities in these protocols may lead to security breaches or even complete network failures. Model checking[15] has proven to be an effective technique for finding such bugs and vulnerabilities. However, since exhaustive model checking of protocol implementations is usually not feasible,[27] model checking is usually applied to models that have been handcrafted starting from protocol standards. This means that model checking is unable to catch bugs that arise because implementations do not conform to their specification. Model learning turns out to be effective in finding exactly this type of bugs, which makes the technique complementary to model checking.

De Ruiter and Poll,[34] for instance, analyzed both server- and client-side implementations of the TLS protocol with a test harness that supported several key exchange algorithms and the option of client certificate authentication. They showed that model learning (or protocol state fuzzing, as they call it) can catch an interesting class of implementation flaws that is apparently common in security protocol implementations: in three out of nine tested TLS implementations new security flaws were found. For the Java Secure Socket Extension, for instance, a model was learned for Java version 1.8.0.25. The authors observed that the model contained *two* paths leading to the exchange of application data: the regular TLS protocol run and another unexpected run. By exploiting this behavior, an attack was possible in which both the client and the server application would think they were talking on a secure connection, where in reality anyone on the line could read the client's data and tamper with it. A fix was released as part of a critical security update, and by learning a model of JSSE version 1.8.0.31, the authors were able to confirm that indeed the problem was solved. Due to a manually constructed abstraction/mapper, the learned Mealy machines were all quite small, with 6–16 states. As the analysis of different TLS implementations resulted in different and unique Mealy machines for each one, model learning could also be used for fingerprinting TLS implementations.

Fiterău et al.[17] combined model learning and model checking in a case study involving Linux, Windows, and FreeBSD implementations of TCP servers and clients. Model learning was used to infer models of different components and then model checking was applied to fully explore what may happen when these components (for example a Linux client and a Windows server) interact. The case study revealed several instances in which TCP implementations do not conform to their RFC specification, see Figure 8 for an example.

Legacy software. Legacy systems have been defined as "large software systems

Figure 7. State machine of SecureCode Aut **application on Dutch Rabo bank card (diagram courtesy of Aarts et al.[3]).**

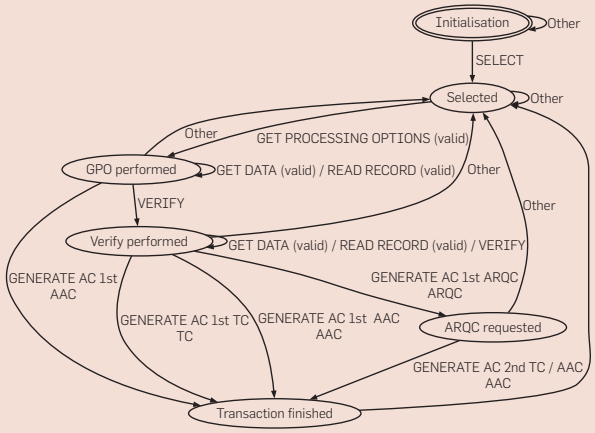
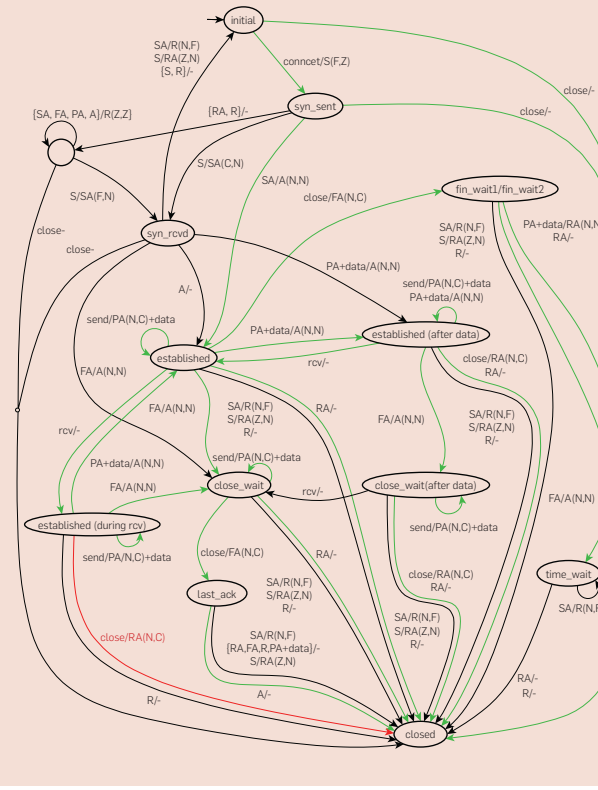


Figure 8. Learned state machine for Windows8 TCP Client (picture courtesy of Fiterau-Brostean et al.[17]). Transitions that are reachable when the Windows8 client interacts with a Windows8 server in a setting with reliable communication are colored green (as computed by a model checker). The red transition marks a nonconformance to the RFC: a CLOSE can generate a RST instead of a FIN even in cases where there is no data to be received, namely, in states where a rcv call is pending.



that we do not know how to cope with but that are vital to our organization."[7] Typically, these systems are based on obsolete technologies, documentation is limited, and the original developers are no longer available. In addition, existing regression tests will be limited. Given these characteristics, innovations that require changes of legacy components are risky. Several techniques have been developed to extract the crucial business information hidden in legacy components, and to support the construction of refactored implementations. Margaria et al.[30] were the first to point out that model learning may help to increase confidence that a legacy component and a refactored implementation have the same behavior.

Schuts et al.,[35] for instance, used model learning to support the rejuvenation of legacy embedded software in a development project at Philips. The project concerned the introduction of a new hardware component, the Power Control Component (PCC), which is used to start-up and shutdown an interventional radiology system. All computers in the system have a software component, the Power Control Service (PCS) which communicates with the PCC over an internal control network during the execution of start-up and shutdown scenarios. To deal with the new hardware of the PCC, which has a different interface, a new implementation of the PCS was needed. Since different configurations had to be supported, with old and new PCC hardware, the old and new PCS software needed to have exactly the same external behavior. Figure 9 illustrates the approach that was followed. From both the legacy implementation A and the refactored implementation B, Mealy machine models $\mathcal{M}_A$ resp. $\mathcal{M}_B$ were obtained using model learning. These models were then compared using an equivalence checker. When the equivalence checker found a counterexample σ, then we checked whether A and $\mathcal{M}_A$ behaved the same on input σ and whether B and $\mathcal{M}_B$ behaved the same on input σ. If there was a discrepancy between A and $\mathcal{M}_A$, or between B and $\mathcal{M}_B$, then we asked the learner to construct an improved model based on counterexample σ. Otherwise σ exhibited a difference between A and B, and we changed either A or B (or both), depending on which response to σ was considered unsatisfactory behavior. The implementations were learned and checked iteratively with increasing sets of stimuli to handle scalability. Issues were found in both the refactored and the legacy implementation in an early stage, before the component was integrated. In this way, costly rework in a later phase of the development was avoided.

Figure 9. Approach to compare legacy component and refactored implementation (diagram courtesy of Schuts et al.[35]).

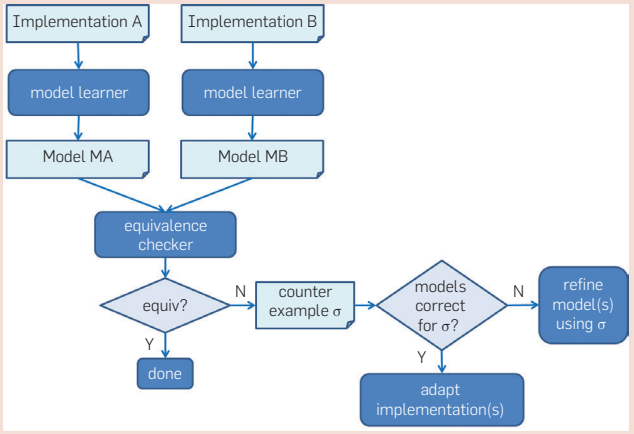
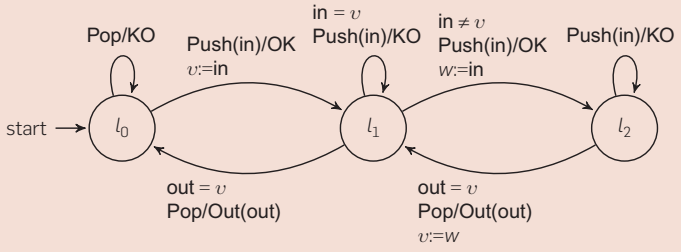


Figure 10. A register automaton.



Recent Advances

During recent years significant progress has been made on algorithms for model learning, which is crucial for scaling the application of these techniques to larger systems.

Basic algorithms. Since 1987, the $L^\star$ algorithm of Angluin's[6] has been considerably improved. The original $L^\star$ performs a membership query for each entry in the observation table. This is often redundant, given that the sole purpose of membership queries is the distinction of states (rows). Therefore, Kearns and Vazirani[28] replaced the observation table of the $L^\star$ algorithm by the so-called discrimination trees, which are basically decision trees for determining equivalence of states.

Another inefficiency of $L^\star$ is that all prefixes of a counterexample are added as rows to the table. Counterexamples obtained through conformance testing or runtime monitoring may be extremely long and are rarely minimal, which results in numerous redundant membership queries. Rivest and Schapire[33] observed that, instead of adding all prefixes of a counterexample as rows to the table, it suffices to add a single, well-chosen suffix as a column.

The new TTT algorithm of Isberner et al.[24, 25] is currently the most efficient algorithm for active learning. The algorithm builds on the ideas of Kearns and Vazirani[28] and Rivest and Schapire[33] but eliminates overly long discrimination trees, which may arise when processing long counterexamples, by cleaning up the internal data structures and reorganizing the discrimination tree. Suppose that a Mealy machine $\mathcal{M}$ has n states and k inputs, and that the length of the longest counterexample returned by the teacher is m. Then in the worst-case TTT requires $O(n)$ equivalence queries and $O(kn^2 +$

n log m) membership queries, each of length $O(n + m)$. This worst-case query and symbol complexity coincides with the algorithm of Rivest and Schapire,[33] but TTT is faster in practice.

The TTT algorithm typically generates more intermediate hypotheses than the L^* algorithm. This suggests that the number of input symbols used in membership queries alone may not be an appropriate metric for comparing learning algorithms: we also need to take into account the number of test queries required to implement equivalence queries. The total number of input symbols in membership and test queries appears to be a sensible metric to compare learning approaches in practice. Two of my students, J. Moerman and A. Fedotov, compared different combinations of learning and testing algorithms on a large number of benchmarks (protocols, control software, circuits, etc.) and found that TTT used on average 3.9 times fewer input symbols than L^*.

Learning and testing can be easily parallelized when it is possible to run multiple instances of the SUL concurrently. Another technique that may speedup learning is to save and restore software states of the SUL (checkpointing). The benefit is that when the learner wants to explore different outgoing transitions from a saved state q it only needs to restore q, which usually is much faster than resetting the system and bringing it back to q via a sequence of inputs. Henrix[21] reports on experiments in which checkpointing with DMTCP speeds up learning with a factor 1.7.

Register automata. Even though we have seen much progress on basic algorithms for learning state machines, these algorithms only succeed to learn relatively small state machines. In order to scale the application of these algorithms to realistic applications, users typically need to manually construct abstractions or mappers.[2] This can be a time-consuming activity that requires several iterations and expert knowledge of the SUL. Therefore, much work has been carried out recently to generalize learning algorithms to richer classes of models that have more structure, in particular EFSM models in which data values may be communicated, stored, and manipulated.

One particular extension for which model learning algorithms have been developed is that of *register automata*.[11] These automata have a finite set of states but are extended with a set of registers that can be used to store data values. Input and output actions are parameterized by data values, which may be tested for equality in transition guards and stored in registers. Figure 10 gives a simple example of a register automaton, a FIFO-set with capacity two. A FIFO-set corresponds to a queue in which only different values can be stored. There is a Push(d) input symbol that tries to insert a value d in the queue, and a Pop input symbol that tries to retrieve a value from the queue. The output in response to a Push is OK if the input value can be added successfully, or KO if the input value is already in the queue or if the queue is full. The output in response to a Pop is Out, with as parameter the oldest value from the queue, or KO if the queue is empty.

Abstraction is the key for scaling model learning methods to realistic applications.

In register automata all data values are fully symmetric, and this symmetry may be exploited during learning. Two different approaches have been explored in the literature. A first approach, followed by Cassel et al.,[12] has been implemented in the software tools LearnLib[26] and RALib.[10] Model learning algorithms usually rely on the Nerode relation for identifying the states and transitions of a learned automaton: two words lead to the same state if their residual languages coincide. The basic idea now is to formulate a Nerode-like congruence for register automata, which determines the states, transitions, and registers of the inferred automaton. Technical basis of the implementation are the so-called symbolic decision trees, which can be used to summarize the results of many tests using a concise symbolic representation.

A second approach for learning register automata, followed by Aarts et al.[1] has been implemented in the software tool Tomte. In this approach, counterexample-guided abstraction refinement is used to automatically construct an appropriate mapper. The idea is to start with a drastic abstraction that completely ignores the data values that occur in input and output actions. When this abstraction is too coarse, the learner will observe nondeterministic behavior. In the example of Figure 10, for instance, an input

sequence Push Push Pop Pop will mostly trigger outputs OK OK Out KO, but sometimes OK OK Out Out. Analysis of this behavior will then lead to a refinement of the abstraction. In our example, for instance, we need at least two abstract versions of the second Push, since apparently it matters whether or not the data value of this input is equal to the data value of the first Push. RALib and Tomte both outperform LearnLib. The performance of Tomte and RALib is roughly comparable. RALib outperforms Tomte on some benchmarks, but Tomte is able to learn some register automata that RALib cannot handle, such as a FIFO-set with capacity 40.

Research Challenges

Even though model learning has been applied successfully in several domains, the field is still in its infancy. There is a huge potential for applications, especially in the area of legacy control software, but more research on algorithms and tools is needed to bring model learning from the current level of academic prototypes to that of an off-the-shelf technology that can be easily applied to a large class of systems. Here, I discuss some of the major research challenges.

Predicates and operations on data. The recent extension of model learning algorithms to register automata is a breakthrough which, potentially, makes model learning applicable to a much larger class of systems. Due to the restriction that no operations on data are allowed, the class of systems that can be described as register automata is small, and mainly consists of academic examples such as the bounded retransmission protocol and some simple data structures. However, as pointed out by Cassel et al.,[12] using SMT solving the new learning algorithms for register automata can be extended to EFSM formalisms in which guards may contain predicates such as the successor and less than relation. A prototype implementation RALib is available and we are close to the point where we can learn models of real-world protocols such as TCP, SIP, SSH, and TLS automatically, without the need to manually define abstractions. Nevertheless, our understanding of algorithms for learning EFSMs with different predicates and operations is still limited, and there are many open questions.

Even though model learning has been applied successfully in several domains, the field is still in its infancy.

Isberner[24] developed a model learning algorithm for *visibly pushdown automata (VPAs)*, a restricted class of pushdown automata proposed by Alur and Madhusudan.[5] This result is in a sense orthogonal to the results on learning register automata: using register automata learning, a stack with a finite capacity storing values from an infinite domain can be learned, whereas using VPA learning it is possible to learn a stack with unbounded capacity storing data values from a finite domain. From a practical perspective it would be useful to develop a learning algorithm for a class of models that generalizes both register automata and VPAs. There are many protocols in which messages may be buffered, and we therefore need algorithms that can learn queues with unbounded capacity.

Beyond Mealy machines. In a Mealy machine, a single input always triggers a single output. In practice, however, a system may respond to an input with zero or more outputs. Moreover, the behavior of systems is often timing dependent and a certain output may only occur if some input has not been offered for a certain amount of time. As a consequence, practical application of model learning is often severely restricted by the lack of expressivity of Mealy machines. For instance, in order to squeeze TCP implementations into a Mealy machine, we had to eliminate timing-based behavior as well as retransmissions.[17] There has been some preliminary work on extending learning algorithms to I/O automata[4] and to event-recording automata,[18] but a major effort is still required to turn these ideas into practical tools.

Systems are often nondeterministic, in the sense that a sequence of inputs may lead to different output events in different runs. Existing model learning tools, however, are only able to learn deterministic Mealy machines. In applications, we can sometimes eliminate nondeterminism by abstracting different concrete output events into a single abstract output, but in many cases this is not possible. Volpato and Tretmans[38] present an adaptation of L^* for active learning of nondeterministic I/O automata. Their algorithm enables learning of nondeterministic SULs, and it allows us to construct partial or approximate

models. Again, a major effort will be required to incorporate these ideas in state-of-the-art tools such as LearnLib, libalf, RALib, or Tomte.

Quality of models. Since the models produced by model learning algorithms have been obtained through a finite number of tests, we can never be sure that they are correct. Nevertheless, from a practical perspective, we would like to be able to make quantitative statements about the quality of learned models and, for instance, assert that a hypothesis is approximately correct with high probability. Angluin[6] proposed such a setting, along the lines of the PAC learning approach of Valiant.[37] Her idea was to assume some (unknown) probability distribution on the set of words over the input alphabet I. In order to test a hypothesis, the conformance tester (see Figure 4) selects a specified number of input words (these are statistically independent events) and checks for each word whether the resulting output of SUL and hypothesis agrees. Only when there is full agreement the conformance tester returns answer *yes* to the learner. An hypothesis is said to be an ϵ-approximation of the SUL if the probability of selecting a string that exhibits a difference is at most ϵ. Given a bound on the number of states of the SUL, and two constants ϵ and δ, Angluin's polynomial algorithm produces a model such that the probability that this model is an ϵ-approximation of the SUL is at least $1 - \delta$. Angluin's result is elegant but not realistic in a setting of reactive systems, since there we typically do not have a fixed distribution over the input words. (Inputs are under the control of the environment of the SUL, and this environment may change.)

Using traditional conformance testing,[29] we can devise a test suite that can guarantee the correctness of a learned model, given an upper bound on the number of states of the SUL. But such an approach is also not satisfactory, since the required number of test sequences grows exponentially with the number of states of the SUL. The challenge therefore is to establish a middle ground between Angluin's approach and traditional conformance testing. Systems logs often provide a probability distribution on the set of input words that may be used as a starting point for defining a metric.

Opening the box. There can be many reasons for using black box model learning techniques. For instance, we may want to understand the behavior of a component but do not have access to the code. Or we may have access to the code but not to adequate tools for analyzing it (for example, in the case of legacy software). Even in "white box" situations where we have access both to the code and to powerful code analysis tools, black box learning can make sense, for instance because a black box model can be used to generate regression tests, for checking conformance to a standard, or as part of model-based development of a larger system. An important research challenge is to combine black box and white box model extraction techniques and, for instance, to use white box methods such as static analysis and concolic testing to help answering equivalence queries posed by a black box learner.

Acknowledgments. Portions of this work were performed in the context of STW projects 11763 (ITALIA) and 13859 (SUMBAT), and NWO projects 628.001.009 (LEMMA) and 612.001.216 (ALSEP). C

References

1. Aarts, F., Fiterău-Broştean, P., Kuppens, H., Vaandrager, F. Learning register automata with fresh value generation. In *ICTAC'15*, LNCS 9399 (2015). Springer, 165–183.
2. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods Syst. Des. 46*, 1 (2015), 1–41.
3. Aarts, F., de Ruiter, J., Poll, E. Formal models of bank cards for free. In *SECTEST'13* (2013). IEEE, 461–468.
4. Aarts, F., Vaandrager, F. Learning I/O automata. In *CONCUR'10*, LNCS 6269 (2010). Springer, 71–85.
5. Alur, R., Madhusudan, P. Visibly pushdown languages. In *STOC'04* (2004). ACM, 202–211.
6. Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.
7. Bennett, K. Legacy systems: coping with success. *IEEE Softw. 12*, 1 (1995), 19–23.
8. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B. On the correspondence between conformance testing and regular inference. In *FASE'05*, LNCS 3442 (2005). Springer, 175–189.
9. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D. libalf: The automata learning framework. In *CAV'10*, LNCS 6174 (2010). Springer, 360–364.
10. Cassel, S., Howar, F., Jonsson, B. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS 15* (2015).
11. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program. 84*, 1 (2015), 54–66.
12. Cassel, S., Howar, F., Jonsson, B., Steffen, B. Active learning for extended finite state machines. *Formal Asp. Comput. 28*, 2 (2016), 233–263.
13. Chalupar, G., Peherstorfer, S., Poll, E., Ruiter, J. Automated reverse engineering using Lego. In *WOOT'14* (Aug. 2014). IEEE Computer Society.
14. Cho, C., Babic, D., Shin, E., Song, D. Inference and analysis of formal models of botnet command and control protocols. In *CCS'10* (2010). ACM, 426–439.
15. Clarke, E., Grumberg, O., Peled, D. *Model Checking*. MIT Press, Cambridge, MA, 1999.
16. Feng, L., Lundmark, S., Meinke, K., Niu, F., Sindhu, M., Wong, P. Case studies in learning-based testing. In *ICTSS'13*, LNCS 8254 (2013). Springer, 164–179.
17. Fiterău-Broştean, P., Janssen, R., Vaandrager, F. Combining model learning and model checking to analyze TCP implementations. In *CAV'16*, LNCS 9780 (2016). Springer, 454–471.
18. Grinchtein, O., Jonsson, B., Leucker, M. Learning of event-recording automata. *Theor. Comput. Sci. 411*, 47 (2010), 4029–4054.
19. Groce, A., Peled, D., Yannakakis, M. Adaptive model checking. *Logic J. IGPL 14*, 5 (2006), 729–744.
20. Hagerer, A., Hungar, H., Niese, O., Steffen, B. Model generation by moderated regular extrapolation. In *FASE'02*, LNCS 2306 (2002). Springer, 80–95.
21. Henrix, M. Performance improvement in automata learning. Master thesis, Radboud University (2015).
22. Hungar, H., Niese, O., Steffen, B. Domain-specific optimization in automata learning. In *CAV 2003*, LNCS 2725 (2003). Springer, 315–327.
23. de la Higuera, C. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
24. Isberner, M. Foundations of active automata learning: An algorithmic perspective. PhD thesis, Technical University of Dortmund (2015).
25. Isberner, M., Howar, F., Steffen, B. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV'14*, LNCS 8734 (2014). Springer, 307–322.
26. Isberner, M., Howar, F., Steffen, B. The open-source LearnLib – A framework for active automata learning. In *CAV'15*, LNCS 9206 (2015). Springer, 487–495.
27. Jhala, R., Majumdar, R. Software model checking. *ACM Comput. Surv. 41*, 4 (Oct. 2009), 21:1–21:54.
28. Kearns, M.J., Vazirani, U.V. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
29. Lee, D., Yannakakis, M. Principles and methods of testing finite state machines—A survey. *Proc. IEEE 84*, 8 (1996), 1090–1123.
30. Margaria, T., Niese, O., Raffelt, H., Steffen, B. Efficient test-based model generation for legacy reactive systems. In *HLDVT'04* (2004). IEEE Computer Society, 95–100.
31. Moore, E. Gedanken-experiments on sequential machines. In *Automata Studies*, Annals of Mathematics Studies 34 (1956). Princeton University Press, 129–153.
32. Peled, D., Vardi, M., Yannakakis, M. Black box checking. *J. Autom. Lang. Comb. 7*, 2 (2002), 225–246.
33. Rivest, R.L., Schapire, R.E. Inference of finite automata using homing sequences. *Inf. Comput. 103*, 2 (1993), 299–347.
34. de Ruiter, J., Poll, E. Protocol state fuzzing of TLS implementations. In *USENIX Security'15 (2015)*. USENIX Association, 193–206.
35. Schuts, M., Hooman, J., Vaandrager, F. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *iFM'16*, LNCS 9681 (2016). Springer, 311–325.
36. Shahbaz, M., Groz, R. Analysis and testing of black-box component-based systems by inferring partial models. *Softw. Test. Verif. Reliab. 24*, 4 (2014), 253–288.
37. Valiant, L.G. A theory of the learnable. In *STOC'84* (1984). ACM, 436–445.
38. Volpato, M., Tretmans, J. Approximate active learning of nondeterministic input output transition systems. *Electron. Commun. EASST 72* (2015).
39. Windmüller, S., Neubauer, J., Steffen, B., Howar, F., Bauer, O. Active continuous quality control. In *CBSE'13* (2013). ACM, 111–120.

Frits Vaandrager (F.Vaandrager@cs.ru.nl), Department of Software Science, Institute for Computing and Information Sciences at Radboud University, Nijmegen, The Netherlands.

Watch the author discuss his work in this exclusive *Communications* video. http://cacm.acm.org/videos/model-learning